Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Logik und Theorie diskreter Systeme
Prof. Dr. Dr. h.c. Wolfgang Thomas

Diplomarbeit

Learning Automata for Streaming XML Documents

Daniel Neider
Matrikelnr. 242559

November 2007

Betreuung: Dr. Christof Löding
Prof. Dr. Dr. h.c. Wolfgang Thomas

Zweitgutachter: Prof. Dr. Ir Joost-Pieter Katoen

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 14. November 2007

Acknowledgments

I would like to thank all the people, who have contributed to this thesis both directly and indirectly.

First of all, I would like thank Prof. Wolfgang Thomas for giving me the possibility to write this thesis. His lectures about automata theory have always been a fresh breeze in the sometimes gray computer science studies.

I am also grateful to Prof. Joost-Pieter Katoen for his kind readiness to co-examine this thesis.

Many thanks go to my supervisor Dr. Christof Löding, who suggested this thesis and encouraged me to pursue this topic. I am very grateful for his numerous comments and suggestions, all the helpful discussions and his invaluable feedback. His guidance has always been of great value to my work.

I am deeply indebted to all the people, who reviewed my thesis. Especially, I would like to thank Emmanuel Müller for his careful proofreading, suggestions and support. Special thanks go to Melanie Sapp, who provided helpful corrections on spelling, grammar and style. I also would like to thank René Boulnois and Lars Otten for their comments and corrections.

Finally, I would like to thank all my friends and family, who were always there to pick me up when things went wrong. I am especially grateful to my parents, without whose support and love this thesis would not have been possible.

Contents

List of Figures

List of Algorithms

Acronyms

cf.	Confer
DFA	Deterministic finite automaton
e.g.	For example
eVPA	Extended visibly pushdown automaton
HPC	High performance computing
i.e.	That is
NFA	Nondeterministic finite automaton
MAT	Minimally adequate teacher
PBFS	Parallel breadth-first search
w.l.o.g.	Without loss of generality
w.r.t.	With respect to
VCA	Visibly one-counter automaton
VPA	Visibly pushdown automaton
XML	eXtensible Markup Language

1 Introduction

The *eXtensible Markup Language (XML)* has rapidly become a popular format for data exchange. Its popularity and importance is based on two prominent features. On the one hand, both data and metadata can be stored as a tree like structure within the same document. In fact, an XML document can be seen as a serialization of a tree where so-called tags surround subtrees. On the other hand, it is extensible in the sense that the user can introduce new semantics, i.e. new tags.

For typing of XML documents several *schema languages* have been proposed. The de facto standard is the *Document Type Definition (DTD)*, which is in principle an extended context free grammar. DTDs are commonly used because of their good tradeoff between applicability and complexity even if DTDs have some limits (in contrast to other XML schema languages like *XML Schema*).

Considering XML as a format for data exchange, the validation of streaming XML documents, i.e. the on-the-fly check whether the document satisfies syntactic and semantic constraints imposed by the DTD, is a substantial task. For this task, concepts of automata theory like pushdown automata can be applied. For example a pushdown automaton, which pushes a symbol on the stack or pops one on reading an opening or a closing tag respectively, can be used. It is not hard to verify that such a pushdown automaton can perform the validation in a single pass with memory bounded by the depth of the XML document. Besides the practical contributions also some questions emerge in this context that are mainly of theoretical interest.

One question is whether a DTD can already be validated by using only a fixed amount of memory, which may depend on the DTD but not on the XML document currently processed. It is clear that not every DTD satisfies this property, which justifies referring to the ones that do as *(strongly) recognizable*. The issue of characterizing such DTDs was addressed by Segoufin and Vianu [SV02], who formalized the constant memory constraint by means of deterministic finite automata. Segoufin and Vianu distinguish two different types of validation depending on whether it includes the check for well-formedness of the input. They refer to them as *strong validation* and *validation*.

The strong validation of XML documents includes the check whether the input is a well-formed XML document. In other words, the strong recognizability is concerned with the question when the set of all XML documents valid w.r.t. a given DTD is regular. Segoufin and Vianu were able to show that this is the case if and only if the DTD is non-recursive.

For the validation of XML documents, it is assumed that the input is a well-formed XML document. A finite automaton is required to identify valid and non-valid XML documents correctly but the behavior on non well-formed XML documents is neglected. In contrast to strong validation, the characterization of recognizability is more involved because validation is less restrictive. In fact, Segoufin and Vianu were only able to identify a complete characterization for fully recursive DTDs, i.e. for DTDs where every two tags that lead to recursive tags are mutually recursive. For the remaining DTDs, they were at least able to find a set of necessary conditions.

The task of constructing a finite automaton capable of (strongly) validating a DTD, of course under the assumption that this DTD is (strongly) recognizable, is even more demanding. In the case of non-recursive DTDs, Segoufin and Vianu provided a direct construction for a deterministic finite automaton that performs the validation. They were also able to extend this construction to the fully recursive case. In a subsequent paper Segoufin and Sirangelo [SS07] provided a another construction using a finite group based approach. However, finding a generic construction method that works for arbitrary recognizable DTDs is still an open problem.

In this thesis, we continue at this point. Under the assumption that a DTD is recognizable, we study how an appropriate DFA can be found. Instead of using a direct construction of a finite automaton from a given DTD, we show that the adaptation of learning algorithms is a very reasonable approach. Learning algorithms are particularly useful if an explicit representation of the target language is needed but where it is unknown how such a representation can be directly computed. Besides the opportunity of finding a solution for the construction problem, it is also possible to study the applicability of learning algorithms to not yet considered settings.

The algorithmic learning of formal languages, especially of regular languages, has been studied intensively. Already in 1967, Gold [Gol67] showed that the "in the limit" identification of a finite automaton from samples of a regular language is generally impossible. Furthermore, Gold [Gol78] showed that the question whether there is a finite automaton with n states that agrees with a given finite set of positive and negative samples of a regular language is NP-complete. A first algorithm that constructs a finite automaton from a such a set of samples was suggested by Biermann and Feldman [BF72]. Another method of this kind was proposed by Gracía and Oncina [GO92]. Their regular positive, negative inference algorithm constructs a deterministic finite automaton (not necessarily of minimal size) is compatible with a given finite set of positive and negative samples in polynomial time that.

The algorithmic learning by interaction with an oracle was introduced by Angluin [Ang87]. She showed that a minimal deterministic finite automaton can be learned efficiently from a minimally adequate teacher, which can answer *membership* and *equivalence queries* about a regular language. On a membership query, the teacher has to check whether a given input belongs to the target language. On an equivalence query, it has to check whether a given conjecture is equivalent to the target language and is required to return a counter-example in the case that the conjecture is not equivalent. Angluin's idea is to organize the results of such queries in a two dimensional table, which is then used to define an equivalence relation over this data. This equivalence is refined until the (finite) Nerode congruence is eventually computed.

Some improvements to Angluin's algorithm have been suggested, which organize the learned data more efficiently. An approach proposed by Rivest and

Schapire [RS89] uses a "reduced table" and a more efficient way of processing counter-examples. In contrast, the algorithm suggested by Kearns and Vazirani [KV94] is based on a tree structure to organize the data. Both methods require fewer queries and, hence, have a better runtime complexity. A detailed discussion of both algorithms can be found in Balcázar, Díaz and Gavaldà's survey [BDG97].

Angluin's paper was a milestone in the theory of algorithmic learning of formal languages because it contains two very fundamental concepts: First, her framework of a minimally adequate teacher can easily be shifted to arbitrary classes of formal languages, in which both the membership and the equivalence problem is decidable. This allows the study of algorithmic learning in a unified framework. Second, the generic idea of learning the Nerode congruence as a canonical representation of the target language was used by many other researches in their development of learning algorithms for language classes of greater expressiveness power.

In the context of validating XML documents, learning algorithms for context free languages and their subclasses (we concentrate on languages acceptable by one-counter automata) are of special interest.[1] Using Angluin's framework Berman and Roos [BR87] developed a polynomial time learning algorithm for one-counter acceptable languages. Their algorithms is based on results that the behavior graph, i.e. the graph induced by the Nerode congruence, of context free languages consists of repeating patterns. The idea is that, because of the repeating structure, a finite initial fragment of the behavior graph is sufficient to fully describe the target language. Fahmy and Roos [FR95] use a similar idea (in combination with the real time acceptor synthesis algorithm of Fahmy and Biermann [FB93]) for efficiently learning real time one-counter automata.

Outline

Inspired by the work of Segoufin and Vianu, in this thesis we address the issue of constructing DFAs for validating serializations of trees against DTDs. We distinguish two different serializations: The XML word representation, an XML style serialization where two matching tags surround each subtree, and the parenthesis word representation, which uses only one kind of parenthesis. With the validation task as motivation, our main interest in this thesis is the study of learning algorithms. We develop learning algorithms for two language classes, which have been studied insufficiently so far or for which no learning algorithms is known.

1. The first class is the class of regular languages with "don't cares". Analogous to the idea of Segoufin and Vianu, a regular language with "don't cares" is accepted by a finite automaton if its behavior is correct on all relevant inputs but the behavior on "don't cares" can be arbitrary.

2. The second class is the class of visibly one-counter acceptable languages. In contrast to one-counter automata, the counter of a visibly one-counter automaton is no longer changed freely but modified according to the type of the current input symbol.

[1] For an comprehensive overview of learning context free languages we refer to Lee's survey [Lee96]

We apply the algorithms we develop to learn automata that can perform the validation of either tree representations respectively.

For learning DFAs capable of validating XML word representations we apply our learning algorithms for regular languages with “don’t cares”. In contrast to learning of regular languages, learning of regular languages with “don’t cares” is more involved. Since the behavior on “don’t cares” is arbitrary, there is no longer a unique language that can be learned. In fact, the learning algorithm has to choose a reasonable behavior on “don’t cares”. Because of this, known methods can be only partly applied and new techniques have to be developed.

We apply our learning algorithm for visibly one-counter acceptable languages to the setting of validating parenthesis word representations. It turns out that in this context either a finite automaton or a visibly one-counter automaton can do the validation equivalently. We develop both a polynomial time learning algorithm for visibly one-counter acceptable languages and a transformation from visibly one-counter automata that validate a DTD to finite automata and vice versa. Surprisingly, it turns out that, in contrast to one-counter automata, the learning of visibly one-counter automata is more involved because their stack behavior is determined by the input.

Besides the theoretical work on learning algorithms, this thesis also covers a practical part. In this, we describe our proof-of-concept implementation (a Java based software), which allows to learn DFAs that validate XML word representations. This software realizes our heuristics for learning regular languages with “don’t cares” and our minimally adequate teacher for XML word representations. An analysis of the software’s runs and results on some sample DTDs finishes the practical part.

This thesis is organized as follows. In Chapter 2 we fix our notation, recall some basics of formal language theory and mathematical logic and introduce the automata models used throughout this thesis.

In Chapter 3 we present Angluin’s learning algorithm for regular languages and the regular positive, negative inference method. Both are the basis for our learning algorithms. This chapter is mainly intended to introduce the algorithmic learning of formal languages but also to match the notation of the original papers to ours.

Chapter 4 addresses the validation of XML word representations. After an introduction to the setting, we extend the minimally adequate teacher to the setting of validating XML word representations and show its construction in detail. Moreover, we present heuristic algorithms for learning regular languages with “don’t cares”. As the application of these learning algorithms on the validation of XML word representations is a major part of this thesis, we also present the results of our proof-of-concept implementation in this chapter.

The validation of parenthesis word representations is addressed in Chapter 5. Again, we introduce the setting and present our visibly one-counter learning algorithm as well as its application to the investigated setting. We also prove the termination and a polynomial runtime of the learner for visibly one-counter acceptable languages.

Finally, Chapter 6 concludes and discusses possible future work.

Preliminaries

2

In this chapter, we recall basics and terminology from formal language theory and mathematical logic. Moreover, we introduce the concepts and automata models we use throughout this thesis and fix their notation. This chapter is not intended to be comprehensive. Hence, for introductory readings we refer to [HMU01], [HU79], [Pap95] and [BPSM$^+$06].

2.1 Basics of formal language theory

At first, we recall some basics from formal language theory. An *alphabet* Σ is a finite set of *symbols*. A *word* is a finite sequence of symbols chosen from an alphabet Σ. We then write $w = a_1 \ldots a_n$ where $a_i \in \Sigma$ for all $i = 1, \ldots n$ and say that w has length n or simply write $|w| = n$. To count the occurrences of a symbol $a \in \Sigma$, we write $|w|_a$. The word that is gained from the empty sequence of symbols is called the *empty word* and denoted by ε. In this case, we set $|\varepsilon| = 0$.

Given two words $u = a_1 \ldots a_n$ and $v = b_1 \ldots b_m$, the *concatenation* of u and v is the word $a_1 \ldots a_n b_1 \ldots b_m$. We write $u \cdot v$ or simply uv. For a word w we call u a *prefix* of w if $w = uv$ for a $v \in \Sigma^*$. The set of all prefixes of w is defined by $pref(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* : \ w = uv\}$.

For each alphabet the set of finite words over Σ is denoted by Σ^*. A subset $L \subseteq \Sigma^*$ is called a *language* (over Σ). The *prefix of a language* is defined as the set of all prefixes of its words, i.e.

$$pref(L) = \bigcup_{w \in L} pref(w) \ .$$

Every total order $<$ over the symbols of Σ induces a total order over the set Σ^*. We say that $u = a_1 \ldots a_n \in \Sigma^*$ is *(canonically) smaller* than $w = b_1 \ldots b_m \in \Sigma^*$, denoted by $u < w$, if and only if $n < m$ or $a_i \ldots a_{i-1} = b_1 \ldots b_{i-1}$ and $a_i < b_i$ for an $i \in \{1, \ldots, n\}$. Respectively, $u \leq w$ if and only if $u < w$ or $u = w$.

Finite automata

Definition 2.1. [Nondeterministic finite automaton] A *nondeterministic finite automaton (NFA)* over the alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_{in}, \Delta, F)$ where Q is a finite, nonempty set of states, Σ is an input alphabet, $Q_{in} \subseteq Q$ is a set of initial states, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation and $F \subseteq Q$ is a set of final states.

$\triangleleft$

The *size* of an NFA is defined as the cardinality of its state set Q. We say that there is an a-transition from q to q' if $(q, a, q') \in \Delta$ and write $q \xrightarrow{a}_{\mathcal{A}} q'$. A *run* of $\mathcal{A}$ on a word $w = a_1 \dots a_n \in \Sigma^*$ is a sequence of states $q_0, \dots, q_n \in Q$ where $q_{i-1} \xrightarrow{a}_{\mathcal{A}} q_i$ for $i = 1, \dots, n$. We then write $q_0 \xrightarrow{w}_{\mathcal{A}} q_n$.

A word w is *accepted* by $\mathcal{A}$ if there exists a run $q_0 \xrightarrow{w}_{\mathcal{A}} q_n$ where $q_0 \in Q_{in}$ is an initial state and $q_n \in F$ is a final state. The set of all words accepted by $\mathcal{A}$ is called the *language* of $\mathcal{A}$ and denoted by $L(\mathcal{A})$. A language L is called *NFA-acceptable* or *regular* if $L = L(\mathcal{A})$ for an NFA $\mathcal{A}$.

An NFA $\mathcal{A}$ where every state $q \in Q$ has exactly one outgoing a-transition for all $a \in \Sigma$ and a unique initial state is called a *deterministic finite automaton (DFA)*. In this case, we replace the transition relation by a transition function $\delta : Q \times \Sigma \to Q$ where $\delta(q, a) = q'$ if and only if $(q, a, q') \in \Delta$. Furthermore, we replace the set of initial states by the unique initial state. Note that every NFA can be determinized with at most exponential blowup in the number of states.

It is sometimes useful to leave some transitions of a DFA undefined. We call such DFAs *partial DFAs*. Such DFAs can be easily completed by adding a sink state and directing all undefined transitions to this new sink state.

An equivalence relation $\sim \subseteq Q \times Q$ over the set of states of a DFA $\mathcal{A}$ induces a *quotient automaton* $\mathcal{A}_{/\sim}$. This quotient automaton is gained from $\mathcal{A}$ by merging $\sim$-equivalent states. Formally we define $\mathcal{A}_{/\sim}$ as follows.

Definition 2.2. [Quotient automaton] Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA and $\sim \subseteq Q \times Q$ an equivalence relation over the set of states of $\mathcal{A}$. The *quotient automaton* $\mathcal{A}_{/\sim} = (Q_{/\sim}, \Sigma, Q_{in/\sim}, \Delta_{/\sim}, F_{/\sim})$ is defined by

- $Q_{/\sim}$ is the set of all $\sim$-classes $q_{/\sim}$ for $q \in Q$,
- $Q_{in/\sim} = \{q_{0/\sim}\}$,
- $P \in F_{/\sim}$ if and only if there is a $p \in P$ such that $p \in F$ and
- $\Delta_{/\sim}$ defined by

$$(P, a, R) \in \Delta_{/\sim} \;\Leftrightarrow\; \exists p \in P, r \in R : \; (p, a, r) \in \Delta \; .$$

$\triangleleft$

Remark 2.3. If $\sim$ is a congruence in the sense that for all $p, q \in Q$ and $a \in \Sigma$ the condition

$$p \sim q \;\Rightarrow\; \delta(p, a) \sim \delta(q, a)$$

holds, then $\mathcal{A}_{/\sim}$ is also deterministic.

$\triangleleft$

Since $\mathcal{A}_{/\sim}$ has as most as many states as $\mathcal{A}$, they do not recognize the same language in general. The next lemma shows that the language of $\mathcal{A}$ is always a subset of the language recognized by $\mathcal{A}_{/\sim}$.

Lemma 2.4. *Let $\mathcal{A}$ be a DFA and $\sim$ an equivalence relation over the set of states of $\mathcal{A}$. Then, $L(\mathcal{A}) \subseteq L(\mathcal{A}_{/\sim})$ always holds.*

We briefly sketch the proof of Lemma 2.4. We show by induction that, if $q \xrightarrow{w}_{\mathcal{A}} q'$ holds, then also $q_{/\sim} \xrightarrow{w}_{\mathcal{A}_{/\sim}} q'_{/\sim}$ holds. We use this as follows: If $\mathcal{A}$

accepts w, then there is a run $q_0 \xrightarrow{w}_{\mathcal{A}} q$ with $q \in F$. Thus, we also know that $q_{/\sim} \xrightarrow{w}_{\mathcal{A}_{/\sim}} q'_{/\sim}$ and, since $q'_{/\sim} \in F_{/\sim}$, the quotient automaton accepts w, too. However, $\mathcal{A}_{/\sim}$ may also accept additional words.

In contrast to finite automata, *regular expressions* are a different type of language-defining notation for regular languages, which offer a declarative way to define the words of a language. Regular expressions are inductively build from the atoms ε, $\emptyset$ and a for all $a \in \Sigma$ as well as the unary operator * and the binary operator $\cdot$. As abbreviation, we use $E^?$ instead of $\varepsilon + E$ and E^+ instead of $E + E^*$ where E is an arbitrary regular expression. For a detailed definition and further reading we refer to [HMU01].

Nerode congruence

Another important equivalence relation is defined over the words of Σ^* w.r.t. a language L. This equivalence is of special interest when dealing with automata as language acceptors.

Definition 2.5. [Nerode congruence] Let $L \subseteq \Sigma^*$ be a language. For words $u, v \in \Sigma^*$ we define

$$u \sim_L v \;\Leftrightarrow\; \forall w \in \Sigma^* :\; uw \in L \Leftrightarrow vw \in L \;.$$

◁

In this case, we call u and v *L-equivalent*. Moreover, we denote the equivalence class of u by $[\![u]\!]_L = \{w \in \Sigma^* \mid u \sim_L w\}$. The set of all equivalence classes of a language forms a partition over Σ^*.

The number of L-equivalence classes is denoted by $index(\sim_L)$. In general, there can be infinitely many L-equivalence classes for an arbitrary language $L \subseteq \Sigma^*$. In fact, Nerode showed that a language is regular if and only if it has only finitely many Nerode equivalence classes.

Remark 2.6. The equivalence relation $\sim_L$ is a congruence, i.e.

$$u \sim_L v \;\Rightarrow\; ua \sim_L va$$

◁

We can use the Nerode equivalence to construct an acceptor for the language L. Thereto, we use the equivalence classes as states and exploit the congruence property to define the transitions. This is formalized in the following definition.

Definition 2.7. [Behavior graph of a language] For a given language $L \subseteq \Sigma^*$, the *behavior graph* of L is a tuple $BG_L = (Q_L, \Sigma, q_0^L, \delta_L, F_L)$ where

- $Q_L = \{[\![w]\!]_L \mid w \in \Sigma^*\}$ is the set of states,
- $q_0^L = [\![\varepsilon]\!]_L$ is the initial state,
- $\delta_L([\![w]\!]_L, a) = [\![wa]\!]_L$ for $[\![w]\!]_L \in Q_L$, $a \in \Sigma$ is the transition function and
- $F_L = \{[\![w]\!]_L \mid w \in L\}$ is the set of final states.

◁

```
<bibliography>
   <book>
      <title> Introduction to Automata Theory </title>
      <authors>
         <author> John E. Hopcroft </author>
         <author> Rajeev Motwani </author>
         <author> Jeffrey D. Ullman </author>
      </authors>
      <publisher> Addison-Wesley </publisher>
   </book>
   <paper>
      <title> XML Everywhere </title>
      <authors>
         ...
      </authors>
      <conference> Principles of Databases </conference>
   </paper>
   ...
</bibliography>
```

Figure 2.1: XML document of a bibliography database

Because $\sim_L$ is a congruence, the transition function and the set of final states are well defined. Runs, acceptance and the language $L(BG_L)$ of behavior graphs are defined analogously to NFAs. A simple induction over the length of words shows that $L(BG_L) = L$.

It is well known that this construction yields a minimal DFA if used for a regular language L. Since the minimal DFA of a regular language is unique, we call it the *canonical DFA for L*.

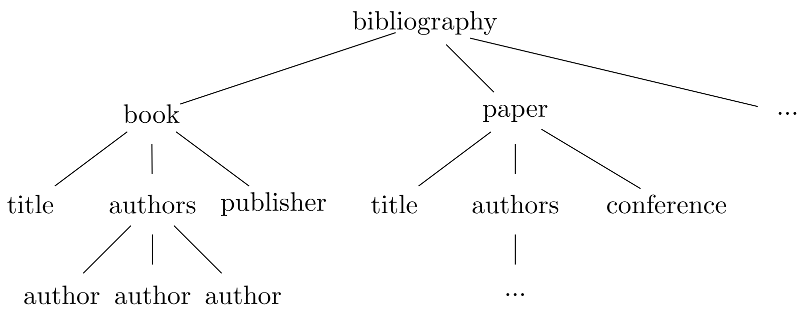
Definition 2.8. [Canonical DFA] Let $L \subseteq \Sigma^*$ be a regular language. The behavior graph of L is called the *canonical DFA for L* and denoted by $\mathcal{A}_L = (Q_L, \Sigma, q_0^L, \delta_L, F_L)$.

$\triangleleft$

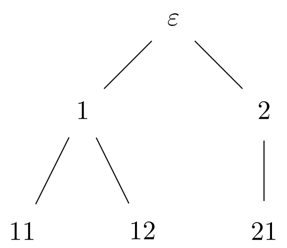
2.2 XML Documents and DTDs

XML documents are often used to store so-called semi-structured data and provide a natural representation of hierarchical and repeating structures. This is achieved by storing the data hierarchically in a tree like structure. Opening and closing tags surround each subtree.

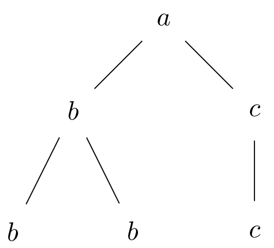
An example of an XML document is depicted in Figure 2.1. It shows an XML document, which stores a bibliography database. Each bibliography consists of a sequence of books and papers, which have a nonempty list of authors. Additionally, each book has a publisher while each paper is presented at a conference. Note that each XML document has a unique root node and that the children of each node are ordered.



(a) The Σ_{tag}-valued tree of the XML document from Figure 2.1



(b) The tree domain dom_t of the tree of Example 2.10



(c) The tree t of Example 2.10

Figure 2.2: Σ-valued trees and tree domains

XML documents can have more features than shown in this small example. Amongst others, it is possible to add attributes to each tag, define namespaces or to use references to other tags and documents. However, in the following we do not consider these more advanced features but they may be worth further investigation.

In fact, XML documents are representations of finite unranked trees over an alphabet Σ_{tag}, whose symbols are called *tags*. Figure 2.2(a) shows the tree corresponding to the XML document in Figure 2.1. In this case, the alphabet used is:

$$\begin{aligned}\Sigma_{tag} \quad = \quad & \{\text{bibliography, book, paper, authors, author,} \\ & \text{title, publisher, conference}\}\end{aligned}$$

As XML documents can have many advanced features, we restrict ourselves to basic concepts in this thesis. To be precise, we only consider the structure of trees but no actual user data or any additional information. However, to be able to define XML documents properly, we first need to define trees and fix some notation.

Definition 2.9. [Σ-valued tree] A Σ-*valued tree* (or simply *tree*) is a pair $t = (dom_t, val_t)$ where $dom_t \subseteq \mathbb{N}_+^*$ is a nonempty, finite tree domain and $val_t : dom_t \to \Sigma$ is a labeling function.

A tree domain has to satisfy the following conditions:

1. dom_t is closed under prefixes
$$ui \in dom_t \;\Rightarrow\; u \in dom_t \;.$$

2. dom_t is closed under "left brothers"
$$ui \in dom_t \;\Rightarrow\; uj \in dom_t \text{ for all } j = 1, \ldots, i-1 \;.$$

◁

The set of all *children* of a node $w \in dom_t$ is denoted by $children_t(w) = \{wi \in dom_t \mid i \in \mathbb{N}\}$; a node $wi \in children_t(w)$ is called a *son* of w while w is called the *father* of wi. Moreover, we call wi and wj *brothers*. The *rank* of a node w is defined by $rank_t(w) = \max_{i \in \mathbb{N}_+} \{wi \in dom_t\}$.

Example 2.10. Let $\Sigma = \{a, b, c\}$. A Σ-valued tree is shown in Figure 2.2(c). Its tree domain is
$$dom_t = \{\varepsilon, 1, 2, 11, 12, 21\} \;.$$
The tree domain is depicted in Figure 2.2(b). The mapping val_t is defined by:
$$\begin{array}{lll} val_t(\varepsilon) = a & val_t(1) = b & val_t(2) = c \\ val_t(11) = b & val_t(12) = b & val_t(21) = c \end{array}$$

◁

Word representations of Σ-valued trees

To be able to validate trees by means of finite automata, they have to be linearized in some way. This is done by transforming a Σ_{tag}-labeled tree into a unique sequence of symbols. However, there exist various such methods.

A very popular method is to use XML documents or *XML word representations* as we call them. Thereby, a Σ_{tag}-valued tree is transformed into a sequence of opening and closing tags. An opening tag is represented by a symbol $a \in \Sigma_{tag}$ while the corresponding closing tag is represented by a symbol $\bar{a}$. The set of all closing tags is denoted by $\bar{\Sigma}_{tag} = \{\bar{a} \mid a \in \Sigma_{tag}\}$. One can think of Σ_{tag} and $\bar{\Sigma}_{tag}$ as different opening and closing parenthesis. Then, the XML word representation of a tree is a balanced word over $\Sigma_{tag} \cup \bar{\Sigma}_{tag}$ corresponding to its depth-first traversal.

Definition 2.11. [XML word representation] For a Σ_{tag}-valued tree t the *XML word representation* $[t]$ over $\Sigma_{tag} \cup \bar{\Sigma}_{tag}$ is defined inductively as

- $[t] = a\overline{a}$ if t is a single root node labeled with a and
- $[t] = a[t_1] \ldots [t_n]\overline{a}$ if t has a root node labeled with a and the subtrees $t_1, \ldots, t_n$.

◁

For a set T of Σ_{tag}-labeled trees, we denote by L_T^{XML} the set of all XML word representations of the Σ_{tag}-labeled trees in T. Furthermore, the set of all XML word representations over Σ_{tag} is denoted by L^{XML} (the alphabet Σ_{tag}

will be clear from the context). With this notation, the XML word representation of the example from Figure 2.2(c) is $abb\bar{b}b\bar{b}\bar{b}cc\bar{c}\bar{c}\bar{a}$.

A second representation uses only one opening and one closing parenthesis, namely "(" and ")". This representation is called *parenthesis word representation*. It is a balanced word over the alphabet $\Sigma_{tag} \cup \{(,)\}$ defined as follows.

Definition 2.12. [Parenthesis word representation] For a Σ_{tag}-labeled tree t the *parenthesis word representation* $[t]^{()}$ over $\Sigma_{tag} \cup \{(,)\}$ is defined inductively as

- $[t]^{()} = a()$ if t is a single root node labeled with a and
- $[t]^{()} = a([t_1]^{()} \dots [t_n]^{()})$ if t has a root node labeled with a and subtrees $t_1, \dots, t_n$.

$\triangleleft$

The parenthesis word representation of the Σ_{tag}-labeled tree in Example 2.2(c) is $a(b(b()b())c(c()))$.

Analogous to the XML word representation, we denote the set of all parenthesis word representations of a set T of Σ_{tag}-labeled trees as $L_T^{()}$. Furthermore, the set of all parenthesis word representations (over Σ_{tag}) is denoted by $L^{()}$.

DTDs

It is often necessary to impose requirements on the structure of trees and therefore on their word representations. In the example of Figure 2.1 such requirements could be:

- Each bibliography consists of at least one book and one paper.
- Each bibliography entity has a title.
- A paper has to be presented at a conference.
- Exactly one publisher publishes a book.

These requirements describe a tree language over the alphabet Σ_{tag}. Such descriptions of tree languages are called *schemas* and provide a typing mechanism for Σ_{tag}-labeled trees. A Σ_{tag}-labeled tree is said to be *valid* w.r.t. a schema if it belongs to the tree language defined by the schema.[1]

There are several schema languages for XML documents. The de facto standard is the *document type definition (DTD)*. Basically, a DTD is an extended context free grammar over an alphabet Σ_{tag}.[2] The tree language defined by a DTD is the set of all derivation trees of the grammar. DTDs are strong enough to express the most requirements in practice but are still easy to handle and well understood. Therefore, we focus on DTDs as the only schema language used in this diploma thesis.

A DTD that formalizes the requirements made above could look like the one in Figure 2.3. The form of this DTD is as proposed by the W3C [BPSM+06]. We

[1] We give a formal definition of validity in Section 4.1.1

[2] An extended context free grammar is a context free grammar that allows regular expressions over Σ_{tag} on the right hand side of each production

```
<!DOCTYPE BIBLIOGRAPHY [
   <!ELEMENT bibliography (book, paper)+>
   <!ELEMENT book         (title, authors, publisher)>
   <!ELEMENT paper        (title, authors, conference)>
   <!ELEMENT authors      (author+)>
   <!ELEMENT title        (#PCDATA)>
   <!ELEMENT author       (#PCDATA)>
   <!ELEMENT publisher    (#PCDATA)>
   <!ELEMENT conference   (#PCDATA)>
]>
```

Figure 2.3: A bibliography DTD

do not go into the details since we define DTDs slightly different. Nevertheless, some more explanations may be useful.

- The root element of each document has to be bibliography. This is defined by the `!DOCTYPE` statement.
- The symbols of the expressions on the right hand side of the rules can be `|` for choice, `,` for sequence, `*` for arbitrary many occurrences, `+` for one or more occurrences and `?` for one or zero occurrences.
- The `#PCDATA` ("parsed character data") statement stands for arbitrary character sequences which represent the data in the document.

For our purpose it is useful to define DTDs a little bit different using regular expressions as right hand sides of productions. This directly allows us to use the finite automata corresponding to these regular expressions. We also change the `#PCDATA` rules to $a \to \varepsilon$ since we are only interested in the structure of Σ_{tag}-labeled trees.

Definition 2.13. [Document type definition] A *document type definition (DTD)* is a tuple $d = (\Sigma_{tag}, r, \rho)$ with a root symbol $r \in \Sigma_{tag}$ and a mapping ρ, which maps each symbol from Σ_{tag} to a regular expression over Σ_{tag}.

$\triangleleft$

In fact, one can think of a DTD as an alternative notation of an extended context free grammar. The mapping ρ can be interpreted as a set of rules of the form $a \to \rho(a)$. Additionally, the root symbol r corresponds to the start symbol of the grammar.

The regular language defined by a regular expression of a right hand side of a rule is called *horizontal language*. For practical use, it is often useful to represent such a regular expressions by a DFA that recognizes the same language. We call those DFAs *horizontal language DFAs* and denote them by $\mathcal{A}_a = (Q_a, \Sigma_{tag}, q_0^a, \delta_a, F_a)$ for every tag $a \in \Sigma_{tag}$.

Intuitively, a Σ_{tag}-valued tree is called valid w.r.t. a DTD if it is a derivation tree of the DTD. The set of all Σ_{tag}-valued trees satisfying a DTD d is denoted by $SAT(d)$. Furthermore, $L(d)$ is the language of all XML word representations that satisfy d, i.e. $L(d) = \{[t] \mid t \in SAT(d)\}$, while the set of all parenthesis word representations that satisfy d is denoted by $L^{()}(d) = \{[t]^{()} \mid t \in SAT(d)\}$.

$$d = \begin{array}{l} a \to bc \\ b \to b^* \\ c \to \varepsilon | c \end{array}$$

Figure 2.4: An example DTD over the alphabet $\Sigma_{tag} = \{a, b, c\}$ with root a

It is clear that both $L(d)$ and $L^{()}(d)$ are context free languages. The languages $L(d)$ are known as Dyck languages.

An example DTD d is given in Figure 2.4. The tree document in Figure 2.2(c) is valid w.r.t. d. In his thesis we use the more natural notation for DTDs like in Figure 2.4 instead the one of the definition: We write $a \to R_a$ instead of $\rho(a) = R_a$ where R_a denotes the regular expression and the corresponding regular language. The root element is always written first. Additionally, rules of the form $a \to \varepsilon$ are not shown.

In contrast to context free grammars, each symbol $a \in \Sigma_{tag}$ of a DTD has a unique rule $a \to R_a$ associated with it. However, this is no restriction since regular languages are closed under union.

2.3 Visibly pushdown automata

Alur and Madhusudan, e.g. [AM04], have intensively studied visibly pushdown automata and their languages. As a proper subclass of pushdown automata, they allow algorithmic verification of many context free properties.

The difference to pushdown automata is the fact that an visibly pushdown automaton cannot freely choose its stack operations but is forced to push, pop or do no stack operation depending on the current input symbol. This restriction results in many good properties not only w.r.t. closure but also for decision problems. For instance like regular languages, visibly pushdown languages are closed under union, complementation, concatenation and the Kleene closure. Moreover, every nondeterministic visibly pushdown automaton with n states can be determinized resulting in a deterministic visibly pushdown automaton with at most $\mathcal{O}(2^{n^2})$ states.

A *pushdown alphabet* $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_{int})$ is a tuple of three disjoint and finite alphabets: Σ_c is a set of *calls*, Σ_r a set of *returns* and Σ_{int} a set of *internal actions*. For every pushdown alphabet $\tilde{\Sigma}$, let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$.

A *visibly pushdown automaton* is defined over the alphabet $\tilde{\Sigma}$. The type of the input symbol determines which action a visibly pushdown automaton has to perform on the stack: A symbol has to be pushed on the stack after reading a call, while a symbol has to be popped after reading a return. An internal action leaves the stack unchanged.

Definition 2.14. [Visibly pushdown automaton] A *visibly pushdown automaton (VPA)* over $\tilde{\Sigma}$ is a tuple $\mathcal{A} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ where Q is a finite, nonempty set of states, Σ is an input alphabet, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet, $F \subseteq Q$ is a set of final states and $\Delta \subseteq (Q \times \Sigma_c \times Q \times \Gamma) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_{int} \times Q)$ is a finite set of transitions.

◁

The *size* of a VPA $\mathcal{A} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ is defined as the number of its states, i.e. $|Q|$.

We use a special symbol $\perp \notin \Gamma$ to represent the empty stack. Thus, the stack of a VPA is a word σ over $\Gamma^* \cdot \{\perp\}$. By convention, the first symbol of σ is the top of the stack. The empty stack is represented by the word $\perp$.

A transition (q, a, q', γ) with $a \in \Sigma_c$ is a push-transition where on reading a, the stack symbol γ is pushed on the stack and the control state is changed to q'. Similarly, a transition (q, a, γ, q') with $a \in \Sigma_r$ is a pop-transition where γ is popped from the stack and the control state is changed to q' while reading a. Note that no pop-transition can be applied on the empty stack. Finally, an internal transition (q, a, q') with $a \in \Sigma_{int}$ leaves the stack unchanged but changes the control state from q to q'.

A *configuration* of a VPA $\mathcal{A}$ is a pair (q, σ) where $q \in Q$ is a control state and $\sigma \in \Gamma^* \cdot \{\perp\}$ is a stack. There is an a-transition from (q, σ) to (q', σ'), denoted by $(q, \sigma) \xrightarrow{a}_{\mathcal{A}} (q', \sigma')$, if the following is satisfied.

- If $a \in \Sigma_c$, then $\sigma' = \gamma\sigma$ for some $(q, a, q', \gamma) \in \Delta$.
- If $a \in \Sigma_r$, then $\sigma = \gamma\sigma'$ for some $(q, a, \gamma, q') \in \Delta$.
- If $a \in \Sigma_{int}$, then $\sigma = \sigma'$ for some $(q, a, q') \in \Delta$.

Let $w = a_1 \dots a_n$ be a word over Σ^*. A *run* of $\mathcal{A}$ on w is a sequence of configurations $(q_0, \sigma_0), \dots, (q_n, \sigma_n)$ where $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i}_{\mathcal{A}} (q_i, \sigma_i)$ for $i = 1, \dots, n$. We denote this by $(q_0, \sigma_0) \xrightarrow{w}_{\mathcal{A}} (q_n, \sigma_n)$. If there exists a word w such that $(q, \sigma) \xrightarrow{w}_{\mathcal{A}} (q', \sigma')$, then we also write $(q, \sigma) \rightarrow^*_{\mathcal{A}} (q', \sigma')$.

A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if there is a run $(q_0, \sigma_0) \xrightarrow{w}_{\mathcal{A}} (q_n, \sigma_n)$, which starts in a configuration with $q_0 \in Q_{in}$ and $\sigma_0 = \perp$ and ends in a configuration with $q_n \in F$ and $\sigma_n = \perp$. The *language* of $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of all words accepted by $\mathcal{A}$. Finally, a language $L \subseteq \Sigma^*$ is said to be *VPA-acceptable* if there is a VPA $\mathcal{A}$ with $L(\mathcal{A}) = L$. We skip the index $\mathcal{A}$ if it is clear from the context.

A VPA only controls which symbols are pushed on the stack but not when they are pushed or popped. Thus, during the run of a VCA on a word w, the height of the stack is determined by the prefix u of w read so far. This intuition allows us to define the *sign* of a symbol $a \in \Sigma$ as $\chi(a) = 1$ if $a \in \Sigma_c$, $\chi(a) = -1$ if $a \in \Sigma_r$ and $\chi(a) = 0$ if $a \in \Sigma_{int}$. Furthermore, we define the *stack height* of a word $w = a_1 \dots a_n$ as $sh(w) = \sum_{i=i}^{n} \chi(a_i)$ where we set $sh(\varepsilon) = 0$.

We observe that a VPA accepts with the empty stack and, thus, can only accept words $w \in \Sigma^*$ with stack height 0. Furthermore, every prefix $u \in pref(w)$ has to have a nonnegative stack height, i.e. $sh(u) \geq 0$ for all $u \in pref(w)$. It is therefore useful to define the languages

$$L_{\geq 0} = \{w \in \Sigma^* \mid \forall u \in pref(w) : \ sh(u) \geq 0\} \text{ and}$$

$$L_{wm} = \{w \in \Sigma^* \mid sh(w) = 0\} \cap L_{\geq 0} \ .$$

A word $w \in L_{wm}$ is called *well matched*.

Example 2.15. The language L_{wm} can be accepted by a trivial one state VPA that uses its stack only to keep track of the stack height.

◁

In contrast to pushdown automata, a VPA has no access to the top stack symbol while reading a call or an internal action. Only on reading a return, the VPA can choose its action depending on the top stack symbol. It turns out that it is useful to allow a VPA to have access to the top stack symbol at all time. We call such VPAs *extended visibly pushdown automata*.

As this extended automata model can alter the top stack symbol, we have to ensure that it cannot remove the bottom symbol of the stack or put it somewhere on the stack. In other words we have to ensure that the stack always is a word $\sigma \in \Gamma^* \cdot \{\perp\}$.

This can be achieved by using two different types of call transitions depending on whether the stack is empty or not. In the first case, the automaton is only allowed to push a symbol on the stack. In the latter case, the automaton has to push a symbol on the stack but can also alter the old top stack symbol.

The same applies to internal actions. On the empty stack, the automaton is not allowed to change the stack while it can alter the top stack symbol if the stack is not empty.

Definition 2.16. [Extended visibly pushdown automaton] An *extended visibly pushdown automaton (eVPA)* over $\widetilde{\Sigma}$ is a tuple $\mathcal{A} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ where Q is a finite, nonempty set of states, Σ is an input alphabet, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet, $F \subseteq Q$ is a set of final states, $\perp \notin \Gamma$ is the bottom stack symbol and $\Delta = \Delta_c \cup \Delta_r \cup \Delta_{int}$ is a finite set of transitions. Thereby,

$$\begin{aligned}
\Delta_c &\subseteq \quad Q \times \Sigma_c \times \{\perp\} \times (\Gamma \cdot \{\perp\}) \times Q \\
&\quad\; \cup\, Q \times \Sigma_c \times \Gamma \times (\Gamma \cdot \Gamma) \times Q \ , \\[1ex]
\Delta_r &\subseteq \quad Q \times \Sigma_r \times \Gamma \times \{\varepsilon\} \times Q \ , \\[1ex]
\Delta_{int} &\subseteq \quad Q \times \Sigma_{int} \times \{\perp\} \times \{\perp\} \times Q \\
&\quad\; \cup\, Q \times \Sigma_{int} \times \Gamma \times \Gamma \times Q \ .
\end{aligned}$$

◁

A transition $(q, a, \gamma, \gamma''\gamma', q') \in \Delta_c$ is a push-transition where on reading a with γ on top of the stack, the topmost stack symbol is changed to γ', the symbol γ'' is pushed on the stack and the control state is changed to q'. Note that $\gamma = \perp \Leftrightarrow \gamma' = \perp$ holds by definition of Δ_c. Similarly, a transition $(q, a, \gamma, \varepsilon, q')$ with $a \in \Sigma_r$ is a pop-transition where γ is popped from the stack and the control state is changed to q' while reading a. Finally, a transition $(q, a, \gamma, \gamma', q')$ with $a \in \Sigma_{int}$ is an internal-transition. On reading a with γ on top of the stack, the topmost stack symbol is replaced by γ' and the control state is changed to q'. Again, $\gamma = \perp \Leftrightarrow \gamma' = \perp$ holds by definition of Δ_{int}.

Remark 2.17. We observe that the transition relation Δ of an eVPA is a subset of $Q \times \Sigma \times (\Gamma \cup \{\perp\}) \times (\Gamma \cup \{\perp\})^* \times Q$. Therefore, we can write transitions in a generic form, namely $(q, a, \gamma, \beta, q') \in \Delta$ where $|\beta| \leq 2$. In fact, we observe that

- $|\beta| = 2$ if $a \in \Sigma_c$,
- $|\beta| = 1$ if $a \in \Sigma_{int}$ and
- $|\beta| = 0$ if $a \in \Sigma_r$.

◁

Remark 2.17 allows us to easily define a-transitions of an eVPA $\mathcal{A}$. We say that there is an a-transition from (q,σ) to (q',σ'), again denoted by $(q,\sigma) \xrightarrow{a}_{\mathcal{A}} (q',\sigma')$ if $\sigma = \gamma\sigma''$ and $\sigma' = \beta\sigma''$ for a $(q,a,\gamma,\beta,q') \in \Delta$. Furthermore, runs, acceptance and languages of eVPAs are defined as for VPAs.

It is obvious that each VPA can easily be transformed into an eVPA. However, we will show that the converse holds, too. Thereto, we provide a construction that transforms an eVPA $\mathcal{A}$ into a VPA $\mathcal{A}'$ simulating $\mathcal{A}$. The basic idea is to delay the push of a stack symbol for one step and store it in the states of the VPA. Thus, $\mathcal{A}'$ has the ability to access to the top stack symbol and can even alter it. To preserve the visibly property, a new bottom stack symbol $\bot'$ has to be pushed on the stack if it is empty. A return transition is simulated by popping the top stack symbol and storing it in the state. We formalize this intuition next.

Construction 2.18. *Let $\mathcal{A} = (Q,\Sigma,Q_{in},\Gamma,\Delta,F)$ be an eVPA. A VPA $\mathcal{A}' = (Q',\Sigma,Q'_{in},\Gamma',\Delta',F')$ that simulates $\mathcal{A}$ is defined as follows.*

- $Q' = Q \times (\Gamma \cup \{\bot\})$,
- $Q'_{in} = Q_{in} \times \{\bot\}$,
- $\Gamma' = \Gamma \cup \{\bot'\}$,
- $F' = F \times \{\bot\}$ *and*
- Δ' *is defined as:*
  - *For $a \in \Sigma_c$*

    $$(q,a,\gamma,\gamma''\gamma',q') \in \Delta \quad \Leftrightarrow \quad ((q,\gamma),a,(q',\gamma''),\gamma') \in \Delta'$$

    for $\gamma \neq \bot$ and $\gamma' \neq \bot$. Furthermore,

    $$(q,a,\bot,\gamma'\bot,q') \in \Delta \quad \Leftrightarrow \quad ((q,\bot),a,(q',\gamma'),\bot') \in \Delta' \ .$$

  - *For $a \in \Sigma_r$*

    $$\begin{aligned}(q,a,\gamma,\varepsilon,q') \in \Delta \quad \Leftrightarrow \quad & ((q,\gamma),a,\bot',(q',\bot)) \in \Delta' \text{ and} \\ & ((q,\gamma),a,\gamma',(q',\gamma')) \in \Delta' \ .\end{aligned}$$

  - *For $a \in \Sigma_{int}$*

    $$(q,a,\gamma,\gamma',q') \in \Delta \quad \Leftrightarrow \quad ((q,\gamma),a,(q',\gamma')) \in \Delta' \ .$$

A not very difficult but technical induction over the length of a word $w \in \Sigma^*$ shows

$$(q,\bot) \xrightarrow{w}_{\mathcal{A}} (q',\sigma') \ \Leftrightarrow \ ((q,\bot),\bot) \xrightarrow{w}_{\mathcal{A}'} \begin{cases} ((q',\bot),\bot) \\ \text{, if } \sigma' = \bot \\ \\ ((q',\gamma_n),\sigma'') \\ \text{, if } \ \sigma' = \gamma_n \ldots \gamma_1\bot \text{ and} \\ \qquad \sigma'' = \gamma_{n-1} \ldots \gamma_1\bot'\bot \end{cases}$$

Thus, there is a one-to-one relationship between each reachable configuration of $\mathcal{A}$ and $\mathcal{A}'$, which shows that $\mathcal{A}'$ can simulate $\mathcal{A}$ step-by-step.

The reachability problem for eVPAs

During this thesis, it is necessary to solve the *point-to-point reachability problem* for eVPAs, which is the following decision problem:

"Given an eVPA $\mathcal{A}$ and two configurations (q, σ) and (q', σ').
Does $(q, \sigma) \rightarrow^*_{\mathcal{A}} (q', \sigma')$ hold?"

We solve a more general version of the point-to-point reachability problem. For an eVPA $\mathcal{A}$ and a set C of configurations, we define

$$pre^*(C) = \{(q, \sigma) \mid (q, \sigma) \rightarrow^*_{\mathcal{A}} (q', \sigma') \text{ for some } (q', \sigma') \in C\} \ .$$

If we are able to compute $pre^*(C)$ for a given set C of configurations, then we are also able to solve the point-to-point reachability problem. To do so, we simply set $C = \{(q', \sigma')\}$ and compute $pre^*(C)$. If $(q, \sigma) \in pre^*(C)$, we know that (q', σ') is reachable from (q, σ). Thus, our task is to compute the set $pre^*(C)$ for some configuration set C.

Already in 1964, Büchi showed that the reachability problem for pushdown automata is decidable. Furthermore, he showed how to compute the set $pre^*(C)$ for a regular set C. Since eVPAs form a proper subset of pushdown automata, we can apply his approach to our setting. However, we use a much simpler and faster algorithm proposed by Bouajjani, Esparza and Maler [EHRS00].

Bouajjani, Esparza and Maler use a special automaton to represent a regular set, i.e. a regular language, of configurations.

Definition 2.19. [P-automaton] Let $\mathcal{A} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ be an eVPA and C a regular set of configurations. A *P-automaton* for C is an NFA $\mathcal{B} = \mathcal{B}(C) = (Q_{\mathcal{B}}, \Gamma, Q, \Delta_{\mathcal{B}}, F_{\mathcal{B}})$ that accepts from an initial state $q \in Q$ exactly those words $\sigma \in \Gamma^* \cdot \{\bot\}$ such that $(q, \sigma) \in C$.

$\lhd$

Note that the initial states of $\mathcal{B}$ are the states of the eVPA $\mathcal{A}$. Furthermore, we say that $\mathcal{B}$ is *normalized* if there are no transitions to an initial state.

The idea is to transform an initially normalized P-automaton $\mathcal{B}$ into an NFA which accepts $pre^*(C)$. Therefore, suppose that $\mathcal{B}$ accepts a word $\beta\sigma$ from state q with the run $q \xrightarrow{\beta} r \xrightarrow{\sigma} q'$ with $q' \in F_{\mathcal{B}}$. Additionally, suppose that there is a transition $(q'', a, \gamma, \beta, q) \in \Delta$. Then, from state q'' the word $\gamma\sigma$ has to be accepted, too, since it is also in $pre^*(C)$. Therefore, we add a transition (q'', γ, r) to $\Delta_{\mathcal{B}}$.

We repeat this procedure until no more transitions can be added. The resulting NFA is denoted by $\bar{\mathcal{B}}$. Since $\bar{\mathcal{B}}$ is gained from $\mathcal{B}$ by saturating transitions, this algorithm is called *saturation algorithm*. Algorithm 1 shows the saturation algorithm in pseudo code. The following theorem states its correctness and complexity.

Theorem 2.20 (Bouajjani, Esparza and Maler [EHRS00])**.** *Given an eVPA $\mathcal{A} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ and a P-automaton $\mathcal{B} = (Q_{\mathcal{B}}, \Gamma, Q, \Delta_{\mathcal{B}}, F_{\mathcal{B}})$ recognizing a set C of configurations, the saturation algorithm yields an NFA $\bar{\mathcal{B}}$ with*

$$(q, \sigma) \in pre^*(C) \ \Leftrightarrow \ q \xrightarrow{\sigma}_{\bar{\mathcal{B}}} q' \text{ with } q' \in F_{\bar{\mathcal{B}}}$$

in time $\mathcal{O}(|\Delta| \cdot |Q| \cdot |Q_{\mathcal{B}}|^3 \cdot |\Gamma|)$.

Input: A normalized P-automaton $\mathcal{B} = (Q_\mathcal{B}, \Gamma, Q, \Delta_\mathcal{B}, F_\mathcal{B})$ for an eVPA $\mathcal{A} = (Q, \Sigma, Q_{in}\Gamma, \Delta, F)$
Output: The P-automaton $\bar{\mathcal{B}}$

$\mathcal{B}_0 = \mathcal{B}$;
$i := 0$;
repeat
  if $(q'', a, \gamma, \beta, q) \in \Delta$, $q \xrightarrow{\beta}_{\mathcal{B}_i} r$ *and* $(q'', \gamma, r) \notin \Delta_{\mathcal{B}_i}$ **then**
    add (q'', γ, r) to $\mathcal{B}_i$ to obtain $\mathcal{B}_{i+1}$;
    $i = i + 1$;
  end
until *no more transition can be added* ;
$\bar{\mathcal{B}} = \mathcal{B}_i$;
return $\bar{\mathcal{B}}$;

Algorithm 1: The saturation algorithm

Proof of Theorem 2.20. The proof of correctness is a rather involved induction and, therefore, skipped here. One can find it in [EHRS00]. Let us just emphasize that it is crucial to provide a normalized P-automaton since the correctness of the saturation algorithm is based on this property.

However, it is not hard to prove the claimed runtime complexity: To check the condition of the if-clause, we have to consider $|\Delta|$ many transitions. Moreover, we have to find a run $q \xrightarrow{\beta} r$. Since $|\beta| \leq 2$, we have to consider $\mathcal{O}(|Q_\mathcal{B}|^2)$ runs of length less or equal than two. Allover, the check of the if-clause has complexity $\mathcal{O}(|\Delta| \cdot |Q_\mathcal{B}|^2)$.

Since a transition of $\mathcal{B}$ is a tuple of states $q \in Q$, $q' \in Q_\mathcal{B}$ and a symbol of the input alphabet Γ, at most $|Q| \cdot |Q_\mathcal{B}| \cdot |\Gamma|$ transitions can be added to $\mathcal{B}$. All in all we get a runtime complexity of $\mathcal{O}(|\Delta| \cdot |Q| \cdot |Q_\mathcal{B}|^3 \cdot |\Gamma|)$.

□

Let us, finally, consider an example to illustrate the saturation algorithm.

Example 2.21. Let $\Sigma_c = \{c\}$, $\Sigma_r = \{r\}$, $\Sigma_{int} = \{a\}$ and $\Sigma = \{c, a, r\}$. Consider the eVPA $\mathcal{A} = (\{q_0, q_1\}, \Sigma, \{q_0\}, \{X, Y\}, \Delta, \{q_0\})$ with the transitions

$$\Delta = \{(q_0, c, \perp, X\perp, q_0), (q_0, c, X, XX, q_0), (q_0, a, X, Y, q_1), (q_1, r, Y, \varepsilon, q_0)\} .$$

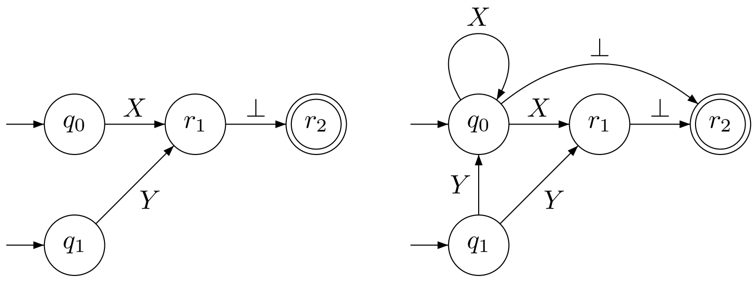
We want to compute $pre^*(C)$ for the set $C = \{(q_0, X\perp), (q_1, Y\perp)\}$. A normalized P automaton $\mathcal{B}$ recognizing C is shown in Figure 2.5(a).

The first transition added to $\mathcal{B}_0$ is $(q_0, \perp, r_2)$ since $(q_0, c, \perp, X\perp, q_0) \in \Delta$ and $q_0 \xrightarrow{X\perp}_{\mathcal{B}_2} r_2$. Then, i is incremented by one.

In the next loop, the transition (q_1, Y, q_0) is added because $(q_1, r, Y, \varepsilon, q_0) \in \Delta$ and $q_0 \xrightarrow{\varepsilon}_{\mathcal{B}_0} q_0$. Again, i is incremented by one.

In the third loop, the transition (q_0, X, q_0) is added to $\mathcal{B}_2$ and i is incremented by one since $q_1 \xrightarrow{Y}_{\mathcal{B}_1} q_0$ and $(q_0, a, X, Y, q_1) \in \Delta$. This is also the last transition that can be added and, thus, $\bar{\mathcal{B}} = \mathcal{B}_3$ is returned. The NFA $\bar{\mathcal{B}}$ is depicted in Figure 2.5(b).

◁



(a) The P-automaton $\mathcal{B}$ (b) The saturated automaton $\bar{\mathcal{B}}$

Figure 2.5: The P-automata used in Example 2.21

Extending the saturation algorithm

The saturation algorithm is an efficient tool to answer the point-to-point reachability problem but does not provide any information about how to reach a configuration from the set C. In many applications this information, i.e. a word w such that $(q, \sigma) \xrightarrow{w} (q', \sigma')$, is also of great importance. Fortunately, we can easily extend the saturation algorithm to provide this information.

The main idea is to store additional information in each transition of the P-automaton added during the run of the saturation algorithm. In particular, for an inserted transition (q'', γ, r) we store which transition $(q'', a, \gamma, \beta, q) \in \Delta$ of the eVPA was used and additionally which prior added transitions of $\mathcal{B}_i$ were used on the run $q \xrightarrow{\beta}_{\mathcal{B}_i} r$. Thus, an accepting run of $\bar{\mathcal{B}}$ starting at state q on a configuration σ yields a sequence of eVPA transitions that can be used to reach a configuration $(q', \sigma') \in C$ from (q, σ). Finally, the sequence of input symbols yield a word w such that $(q, \sigma) \xrightarrow{w} (q', \sigma')$.

We extend the saturation algorithm as follows. Assume that the transition (q'', γ, r) is added in the i-th step because there is an eVPA transition $(q'', a, \gamma, \beta, q) \in \Delta$ and a run $q \xrightarrow{\beta}_{\mathcal{B}_i} r$. Then, the transition (q'', γ, r) is labeled with i and we store both the transition $(q'', a, \gamma, \beta, q)$ and the labels of the transitions that were used in the run $q \xrightarrow{\beta}_{\mathcal{B}_i} r$. Note that only transitions added by the algorithm are labeled and, therefore, only those are stored in a new transition.

Example 2.22. We continue with Example 2.21. The additional information stored for each transition during the run of the saturation algorithm is shown in the table below.

Loop	Added transition	eVPA-transition	Used transitions
1	$(q_0, \perp, r_2)$	$(q_0, c, \perp, X\perp, q_0)$	
2	(q_1, Y, q_0)	$(q_1, r, Y, \varepsilon, q_0)$	
3	(q_0, X, q_0)	(q_0, a, X, Y, q_1)	(2)

◁

The information stored in the transitions of $\bar{\mathcal{B}}$ can be used as follows. Let $q \xrightarrow{\sigma}_{\bar{\mathcal{B}}} q'$ be an accepting run of $\bar{\mathcal{B}}$ on σ starting at state q. Furthermore, suppose that $i_1, \ldots, i_n$ is the sequence of labels of the transitions used on this run. We first observe that there is only additional information for a prefix of σ since the saturation algorithm does only insert transitions starting in states of Q. Moreover, for a transition labeled with i we denote

- the corresponding eVPA transition by $r(i)$ and
- the sequence of labels of transitions used during the insertion of this transition by $\tau(i) = \tau_1^{(i)}, \ldots, \tau_{k_i}^{(i)}$.

We observe that, since the algorithm can only use already existing transitions of the P-automaton, $\tau_j^{(i)} < i$ holds for all $j = 1, \ldots, k_i$. Therefore, we can apply the substitution

$$\lambda(i) = \begin{cases} r(i) & \text{, if } k_i = 0 \\ r(i), \lambda(\tau_1^{(i)}), \ldots, \lambda(\tau^(i)_{k_i}) & \text{, else} \end{cases}$$

to the sequence $i_1, \ldots, i_n$ and gain a sequence $\lambda(i_1), \ldots, \lambda(i_n) = \delta_1, \ldots \delta_m$ of eVPA transitions with $\delta_j = (q_j, a_j, \gamma_j, \beta_j, q'_j) \in \Delta$ for $j = 1, \ldots, m$. This sequence of eVPA transitions can be used to reach a configuration $(q', \sigma') \in C$ from (q, σ). Thus, the word $w = a_1 \ldots a_m$ where a_j is the input symbol of the j-th transition δ_j can be used to reach (q', σ') from (q, σ), i.e. $(q, \sigma) \xrightarrow{w}_{\mathcal{A}} (q', \sigma')$.

Example 2.23. We continue Example 2.22 and compute a word w such that $(q_0, XX\bot) \xrightarrow{w}_{\mathcal{A}} (q', \sigma')$ for a $(q', \sigma') \in C$. Since $(q_0, XX\bot) \in pre^*(C)$, there is an accepting run of $\bar{\mathcal{B}}$ on $XX\bot$ starting at q_0, namely

$$q_0 \xrightarrow{X}_{\bar{\mathcal{B}}} q_0 \xrightarrow{X}_{\bar{\mathcal{B}}} q_0 \xrightarrow{\bot}_{\bar{\mathcal{B}}} r_2 \ .$$

The sequence of labels of the transitions used by this run is 331. By applying the substitution to this sequence, we gain

$$\begin{aligned} \lambda(3), \lambda(3), \lambda(1) &= \delta_3, \lambda(2), \delta_3, \lambda(2), \delta_1 \\ &= \delta_2, \delta_2, \delta_3, \delta_2, \delta_1 \end{aligned}$$

where $\delta_1 = (q_0, c, \bot, X\bot, q_0)$, $\delta_2 = (q_1, r, Y, \varepsilon, q_0)$ and $\delta_3 = (q_0, a, X, Y, q_1)$ as shown in Table 2.22. Finally, we gain $w = ararc$.

$\triangleleft$

We note that w can be computed in time linear in the length of w. However, this may be exponential in $|Q|$ and $|\Gamma|$.

This approach has some weaknesses. On the one hand, we cannot guarantee which configuration $(q', \sigma') \in C$ is reached from (q, σ) with w. Since $\bar{\mathcal{B}}$ is perhaps nondeterministic, there can be several accepting runs. It is, therefore, possible to reach several configurations in C with different transition sequences.

On the other hand, the word w computed may not be the canonically smallest word with the property $(q, \sigma) \xrightarrow{w}_{\mathcal{A}} (q', \sigma')$. For instance in Example 2.23, the run

$$q_0 \xrightarrow{X}_{\bar{\mathcal{B}}} q_0 \xrightarrow{X}_{\bar{\mathcal{B}}} r_1 \xrightarrow{\bot}_{\bar{\mathcal{B}}} r_2$$

is also accepting for $XX\bot$ starting at q_0 but the word ar obtained from this run is canonically smaller than $ararc$.

To compute a smallest word, we use a two-tiered approach. Let $\mathcal{A}$ be an eVPA, (q, σ) a configuration and C a set of configurations of $\mathcal{A}$.

1. First, we use the saturation algorithm to compute $pre^*(C)$. Now, we check whether $(q, \sigma) \in pre^*(C)$ or not. In the latter case we return "no".

2. Now, we compute the sets

$$pre^i(C) = \{(q', \sigma') \mid \exists a \in \Sigma, (q'', \sigma'') \in pre^{i-1}(C) : \ (q', \sigma') \xrightarrow{a}_{\mathcal{A}} (q'', \sigma'')\}$$

   for $i = 1, \ldots$ where $pre^0(C) = C$ until $(q, \sigma) \in pre^n(C)$ is eventually discovered for some $n \geq 0$. Note that the existence of n is guaranteed since we know $(q, \sigma) \in pre^*(C)$. During this procedure, we incrementally compute the smallest word w such that $(q', \sigma') \xrightarrow{w}_{\mathcal{A}} (q'', \sigma'') \in C$ with $(q'', \sigma'') \in C$ for every $(q', \sigma') \in pre^i(C)$. If $(q, \sigma) \in pre^n(C)$ is finally discovered, we return its smallest word.

However, the worst-case complexity of this approach can be exponential in the length of w. Thus, it is doubtful if this is an applicable technique.

Product of an eVPA and a DFA

In a later chapter of this thesis we are interested in the combined behavior of an eVPA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, Q_{in}^{\mathcal{A}}, \Gamma_{\mathcal{A}}, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ and a DFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$, which run in parallel on the same input word $w \in \Sigma^*$. One possibility to study their behavior is to construct a simple product eVPA $\mathcal{A} \otimes \mathcal{B}$. The state space of the product automaton consists of all pairs of states of the automata $\mathcal{A}$ and $\mathcal{B}$. The transitions are defined accordingly to the transitions of both automata. Note that the product construction is well defined because the stack operations are exclusively determined by the eVPA $\mathcal{A}$.

Since we are only interested in the behavior of the parallel run of both automata but not in their accepted languages, the set of final states of the product automaton can be arbitrary. E.g. we can set the final state set to $F_{\mathcal{A}} \times F_{\mathcal{B}}$ to recognize the intersection of $L(\mathcal{A})$ and $L(\mathcal{B})$.

The formal definition of the product of an eVPA and a DFA can be found in Definition 2.24.

Definition 2.24. Let $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, Q_{in}^{\mathcal{A}}, \Gamma_{\mathcal{A}}, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ be an eVPA and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$ a DFA over the same input alphabet Σ. The *product of $\mathcal{A}$ and $\mathcal{B}$*, denoted by $\mathcal{A} \otimes \mathcal{B} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$, is again an eVPA defined as follows.

- $Q = Q_{\mathcal{A}} \times Q_{\mathcal{B}}$,
- $q_0 = Q_{in}^{\mathcal{A}} \times \{q_0^{\mathcal{B}}\}$,
- $\Gamma = \Gamma_{\mathcal{A}}$ and
- Δ defined as

$$((q, p), a, \gamma, \beta, (q', p')) \in \Delta \quad \Leftrightarrow \quad (q, a, \gamma, \beta, q') \in \Delta_{\mathcal{A}} \text{ and } \delta_{\mathcal{B}}(p, a) = p' \ .$$

- F can be an arbitrary subset of Q.

$\triangleleft$

The first component of the state set and the stack of $\mathcal{A} \otimes \mathcal{B}$ are used to simulate the run of eVPA $\mathcal{A}$ on an input word $w \in \Sigma^*$ while the second component of the state set is used to simulate the run of the DFA $\mathcal{B}$ on w. Lemma 2.25 formally states this connection.

Lemma 2.25. *Let $\mathcal{A} \otimes \mathcal{B} = (Q, \Sigma, Q_{in}, \Gamma, \Delta, F)$ be the product of an eVPA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, Q_{in}^{\mathcal{A}}, \Gamma_{\mathcal{A}}, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ and a DFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$ over the same input alphabet Σ. Then,*

$$(q, \sigma) \xrightarrow{w}_{\mathcal{A}} (q', \sigma') \text{ and } p \xrightarrow{w}_{\mathcal{B}} p' \;\Leftrightarrow\; ((q, p), \sigma) \xrightarrow{w}_{\mathcal{A}\otimes\mathcal{B}} ((q', p'), \sigma')$$

holds for each $w \in \Sigma^*$.

The proof of Lemma 2.25 is a straightforward induction over the length of an input $w \in \Sigma^*$ and, therefore, skipped.

Visibly one-counter automata

Visibly one-counter automata are deterministic visibly pushdown automata that have only a single stack symbol so that they can use their stack only as a counter. However, for reasons of simplicity we define them with a counter that can have arbitrary natural numbers. Moreover, we allow that a visibly one-counter automaton can check his counter up to a fixed threshold. Again, the type of the input symbol determines the action performed on the counter, i.e. whether the counter is incremented, decremented or left unchanged. In this context, we speak no longer of the stack height of a word $w \in \Sigma^*$ but of its *counter value* $cv(w) = sh(w)$.

Definition 2.26. [Visibly one-counter automaton] A *visibly one-counter automaton with threshold m (m-VCA)* over a pushdown alphabet $\tilde{\Sigma}$ is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta_0, \ldots, \delta_m, F)$ where Q is a finite, nonempty set of states, Σ is an input alphabet, $q_0 \in Q$ is a initial state, $\delta_i : Q \times \Sigma \to Q$ is a transition function for every $i = 0, \ldots, m$ and $F \subseteq Q$ is a set of final states.

$\lhd$

A *configuration* of $\mathcal{A}$ is a pair (q, k) where $q \in Q$ is a state and $k \in \mathbb{N}$ is a counter value. There is an *a-transition* from (q, k) to (q', k'), denoted by $(q, k) \xrightarrow{a}_{\mathcal{A}} (q', k')$ if $k' = k + \chi(a) \geq 0$ and $\delta_k(q, a) = q'$ if $k < m$ and $\delta_m(q, a) = q'$ if $k \geq m$. Note that the restriction $k' \geq 0$ prevents the application of a return transition while the counter has value 0.

A *run* of $\mathcal{A}$ on a word $w = a_1 \ldots a_n \in \Sigma^*$ is a sequence of configurations $(q_0, k_0), \ldots, (q_n, k_n)$ with $(q_{i-1}, k_{i-1}) \xrightarrow{a_i}_{\mathcal{A}} (q_i, k_i)$ for $i = 1, \ldots, n$. We also write $(q_0, k_0) \xrightarrow{w}_{\mathcal{A}} (q_n, k_n)$. The word w is *accepted* by $\mathcal{A}$ if there is a run $(q_0, 0) \xrightarrow{w}_{\mathcal{A}} (q', 0)$ where $q' \in F$. The *language* of an m-VCA $\mathcal{A}$ is the set of all words accepted by $\mathcal{A}$. A language $L \subseteq \Sigma^*$ is said to be *m-VCA-acceptable* if there is an m-VCA that accepts L. Moreover, we call L *VCA-acceptable* if there is an $m \in \mathbb{N}$ such that L is m-VCA-acceptable.

Every m-VCA induces a configuration graph.

Definition 2.27. [Configuration graph of an m-VCA] Consider an m-VCA $\mathcal{A} = (Q, \Sigma, q_0, \delta_0, \ldots, \delta_m, F)$ over the tag alphabet $\tilde{\Sigma}$. The *configuration graph* $G_{\mathcal{A}} = (V_{\mathcal{A}}, \Sigma, (q_0, 0), E_{\mathcal{A}}, F_{\mathcal{A}})$ of $\mathcal{A}$ is defined as follows:

- $(q_0, 0)$ is the initial configuration.
- $V_\mathcal{A} = \{(p, j) \in Q \times \mathbb{N} \mid \exists w \in \Sigma^* : (q_0, 0) \xrightarrow{w}_\mathcal{A} (p, j)\}$ is the set of all configurations reachable from the initial configuration.
- $E_\mathcal{A} \subseteq V_\mathcal{A} \times \Sigma \times V_\mathcal{A}$ with $((q, i), a, (p, j)) \in E_\mathcal{A}$ if $(q, i) \xrightarrow{a}_\mathcal{A} (p, j)$ are the edges of $G_\mathcal{A}$. We also denote such edges as $(q, i) \xrightarrow{a}_{G_\mathcal{A}} (p, j)$.
- $F_\mathcal{A} = \{(q, 0) \in V_\mathcal{A} \mid q \in F\}$ is the set of final configurations.

$\lhd$

One can think of a configuration graph as an (infinitive) state machine, which simulates the computation of $\mathcal{A}$ and accepts L. It is clear that $(q, i) \xrightarrow{w}_\mathcal{A} (p, j)$ if and only if $(q, i) \xrightarrow{w}_{G_\mathcal{A}} (p, j)$ holds for any reachable configuration (q, i). Since $\mathcal{A}$ is deterministic, the configuration graph is also deterministic in the sense that each state has at most one a-successor for every $a \in \Sigma$.

2.4 Boolean formulae

Next, we recall basic results from mathematical logic and fix some notations. We begin with Boolean formulae.

Definition 2.28. Let $\tau = \{x_1, \ldots, x_n\}$ with $x_i \in \{0, 1\}$ for $i = 1, \ldots, n$ be a finite set of Boolean variables. A *Boolean formula* over the set τ is constructed inductively as follows.

- x is a Boolean formula for all $x \in \tau$. Moreover, the Boolean constants 0 and 1 are Boolean formulae.
- If φ is a Boolean formula, then $\neg\varphi$ is again a Boolean formula.
- If φ and ψ are Boolean formulae, then $(\varphi \wedge \psi)$ is a Boolean formula, too.

$\lhd$

This inductive definition of Boolean formulae ensures that each formula has a unique "evaluation". However, for a better reading we drop the parenthesis when the evaluation is unambiguous. Moreover, note that the logical operators $\neg$ and $\wedge$ are functionally complete and all other logical operators can be defined by only using them.

As a Boolean formula is constructed from a set τ of Boolean variables, we do not require every variable from τ to be used in this formula. Thus, we denote the set of all Boolean variables that are used in φ by $\tau(\varphi)$.

A mapping

$$\mathfrak{I} : \tau(\varphi) \to \{0, 1\}$$

that assigns a Boolean value to each variable of a formula φ is called an *interpretation* of φ. We also call $\mathfrak{I}$ an *assignment* of Boolean values to the formula φ. Every such interpretation defines a Boolean value of a formula φ, which is defined as follows.

Definition 2.29. [Boolean value of a formula] Let φ be a Boolean formula and $\mathfrak{I} : \tau(\varphi) \to \{0, 1\}$ be an interpretation of φ. The *Boolean value* $[\![\varphi]\!]^\mathfrak{I}$ of the formula φ is defined inductively as

- $[\![0]\!]^{\mathfrak{I}} = 0$ and $[\![1]\!]^{\mathfrak{I}} = 1$,
- $[\![x]\!]^{\mathfrak{I}} = \mathfrak{I}(x)$ for all variables $x \in \tau(\varphi)$,
- $[\![\neg\varphi]\!]^{\mathfrak{I}} = 1 - [\![\varphi]\!]^{\mathfrak{I}}$,
- $[\![\varphi \wedge \psi]\!]^{\mathfrak{I}} = \min([\![\varphi]\!]^{\mathfrak{I}}, [\![\psi]\!]^{\mathfrak{I}})$.

$\lhd$

An interpretation $\mathfrak{I}$ of a formula φ with $[\![\varphi]\!]^{\mathfrak{I}} = 1$ is called a *model* of φ. As an abbreviation we also write $\mathfrak{I} \models \varphi$ and use the symbol μ for a model of φ.

The problem of computing a model for a Boolean formula is known as *satisfiability problem* or simply *SAT*. Unfortunately, Cook showed that this problem is NP-complete for arbitrary Boolean formulae [Coo71], [Pap95]. However, up-to-date SAT solvers are capable of solving Boolean formulas with hundreds of thousands of variables within a few seconds or minutes. Thus, we assume that they can be used as subroutines even in large-scale applications.

Each Boolean formula can be transformed into a canonical form, the so-called *conjunctive normal form*. This normal form consists of a conjunction of disjunctions where each negation is only applied to Boolean variables. Thereby, a Boolean variable or its negation is called a *literal*.

Definition 2.30. [Conjunctive normal form] A Boolean formula φ is in *conjunctive normal form (CNF)* if it is of the form

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} Y_{i,j}$$

where $Y_{i,j}$ are literals. Each subformula $\bigvee_{j=1}^{m_i} Y_{i,j}$ is called a *clause*.

$\lhd$

As mentioned above, each Boolean formula can be transformed into CNF by using the distributive and DeMorgan's law. However, in general this results in a formula, which has exponentially many clauses.

Learning regular languages

3

The learning algorithms developed in this thesis are based on two learning algorithms for regular languages. It is, therefore, useful to take a closer look at their function. This chapter is organized as follows. We first give a brief introduction to the theory of learning unknown languages and then present two learning algorithms for regular languages in very detail. The first algorithm, introduced in Section 3.1, is Angluin's learning algorithm [Ang87]. The second one is the regular positive, negative inference method proposed by Gracía and Oncina [GO92]. It is presented in Section 3.2. The proofs of the following sections are suggested by Christof Löding, who did them in his class about Advanced Automata Theory.[1]

The general goal of learning algorithms is to identify an automaton, preferable of minimal size, that agrees with an a priory fixed *target language* $L \subseteq \Sigma^*$ over a known alphabet Σ. In general we can distinguish between two kinds of algorithms, which we call *query* and *non-query* algorithms. Query algorithms can query an oracle for information about words of the target language while non-query algorithms are provided with a fixed set of examples of the target language and have to provide a minimal DFA compatible with the given samples.

The most popular learning framework for query algorithms is the one proposed by Angluin [Ang87]. In this setting the learning algorithm, the so-called *learner*, who knows initially nothing about the target language, tries to learn the target language by means of querying a *teacher*. This teacher is capable of correctly answering two types of queries about the target language L.

- The first type is a *membership query* on a word w. The answer to this query is "yes" or "no" depending on whether $w \in L$ or not.
- The second type is an *equivalence query* on a conjecture M. The teacher answers "yes" if M is a correct description of L. Otherwise the teacher replies with a counter-example w, i.e. w belongs to L if and only if w does not belong to M. The word w is called a counter-example because it is a witness that L and M are distinct. We make no assumptions on the process of generating counter-examples. E.g. they do not need to be the smallest ones.

Note that all teachers for the language L have to give the same answer to membership queries but they may provide different counter-examples on equivalence queries.

[1] In fact, Section 3.1 and Section 3.2 are taken and adapted from Angluin's [Ang87] and Gracía and Oncina's [GO92] paper respectively. This is necessary to fit into our terminology

Angluin's framework is related to experimentation or, in a pictorial language, the process of human learning. It is based on the idea that a student asks his professor about an unknown subject and eventually comes up with a theory. The professor checks this theory and provides a counter-example if necessary. However, it seems to be impossible to define formally what the minimum requirements of a teacher are but the ability to answer membership and equivalence queries seems to be a natural one. Therefore, such teachers are called *minimally adequate teachers (MATs)*. Answering membership questions is an unproblematic capability but answering equivalence queries and finding counter-examples is more demanding because it requires the teacher to have an explicit representation of the target language. Additionally, the equivalence has to be decidable for the considered class of languages. This restricts the domain of application.

We demand that the learning algorithms are efficient, i.e. that the runtime is bounded by a fixed polynomial of two kinds of parameters: The first kind is a measurement for the complexity of the target language. For instance, one could use the size of a minimal automaton recognizing the target language. The second kind is a measurement for the complexity of the answers of the MAT, which could be for instance the length of the longest counter-example. This reflects the fact that a counter-example should be at least read completely. Besides the runtime, a further important aspect is the number of membership and equivalence queries a learner asks. We also require that these quantities are polynomial.

Non-query algorithms like Biermann and Feldman's algorithm [BF72] or the regular positive, negative inference suggested by Gracía and Oncina [GO92] try to infer an automaton that is compatible with a given set of samples. Clearly, if they shall result in an automaton for a target language, then the samples have to contain enough information about the language. Such learners can also be used to actively learn a target language from a MAT if counter-examples on equivalence queries are used to extend the set of samples. An interesting question is whether an equivalent automaton is eventually computed or how the behavior "in the limit" of such learning algorithms is.

3.1 Angluin's learner

The task of learning a regular language $L \subseteq \Sigma^*$ from a MAT is to provide a (minimal) DFA $\mathcal{A}$ with $L(\mathcal{A}) = L$. On a membership query the MAT is faced with a word w and has to decide if $w \in L$. On an equivalence query the MAT is provided with a DFA $\mathcal{A}$ and has to check whether $L(\mathcal{A}) = L$ holds.

The following naive approach can already achieve this learning task: Since the DFAs over a fixed alphabet can be enumerated, ordered by their size, a learning algorithm can successively ask equivalence queries on these sequence until a DFA recognizing L is found. Clearly, this algorithm returns the minimal DFA for the target language. This shows that the learning task for regular languages can be solved in a much simpler framework, which only provides the possibility to ask equivalence queries but does not supply any counter-examples. Since this clearly is an infeasible approach in practice, a learner should make use of membership queries and the presence of counter-examples to enhance its performance.

Angluin [Ang87] proposed an efficient algorithm for learning regular languages from a MAT. In her paper she shows that a minimal DFA for a regular language L can be learned efficiently, i.e. in polynomial time in the size of the minimal DFA recognizing L (which is a measurement for the complexity of the target language) and the length of the longest counter-example returned by the teacher (which somehow measures the inefficiency of the MAT).

Theorem 3.1 (Angluin [Ang87])**.** *Let $L \subseteq \Sigma^*$ be a regular language. Given a minimally adequate teacher for L, the minimal DFA $\mathcal{A}_L$ can be learned in time polynomial in the size of $\mathcal{A}_L$ and the length of the longest counter-example provided by the teacher.*

Angluin's approach is to learn the Nerode congruence of the regular target language L. This is done by approximating the Nerode congruence starting with a coarse partition. During the run of the algorithm, this approximation is refined until the Nerode congruence is eventually computed. Alternatively, one can think of this procedure as the learning of the target language's behavior graph. Note that this procedure terminates because the index of $\sim_L$ is finite for every regular language.

The learner stores the gathered information about the target language in a finite table, the so-called *observation table*. This table also manages the approximation of the Nerode congruence. A formal definition of an observation table is given next.

Definition 3.2. [Observation table] An *observation table* is a tuple $O = (R, S, T)$ consisting of

- a nonempty, finite, prefix-closed set $R \subseteq \Sigma^*$ of *representatives*,
- a nonempty, finite, suffix-closed set $S \subseteq \Sigma^*$ of *samples* and
- a mapping $T : (R \cup R \cdot \Sigma) \cdot S \to \{0, 1\}$.

We call O an *observation table for* $L \subseteq \Sigma^*$ if $T(w) = 1 \Leftrightarrow w \in L$ holds for all w in the domain of T.

$\triangleleft$

If O is an observation table for L, then the data stored in the table agrees with the target language. It is easy for Angluin's learner to maintain this property by properly adding the information obtained from membership queries. Therefore, we do not mention this property explicitly and assume that O is always an observation table for the target language.

One can think of an observation table for a language L as a table of Boolean values. The rows of the observation table are labeled with words of $R \cup R \cdot \Sigma$ and the columns are labeled with words of S. The entry of the row u and the column w is $T(u \cdot w)$. If O is an observation table for L, then the entry $T(u \cdot w)$ is 1 if and only if the word $u \cdot w$ belongs to L. Note that ε is always a representative (because R is prefix closed) and always a sample (since S is suffix closed). Figure 3.1 on page 32 shows examples of observation tables.

It is useful to think of an observation table as such a two-dimensional table even if this representation may shows information redundantly. It is so possible to distinguish two rows by their columns. We say that the rows of $u, v \in R \cup R \cdot \Sigma$ are equal if and only if

$$T(u \cdot w) = T(v \cdot w)$$

holds for all $w \in S$. Analogously, the rows of u and v are not equal if and only if there is a sample $w \in S$ with $T(u \cdot w) \neq T(v \cdot w)$.

As mentioned above, an observation table manages an approximation of the Nerode congruence of the target language. To see this, recall how two non-equivalent L-equivalence classes $[\![u]\!]_L$ and $[\![v]\!]_L$ can be distinguished: There is a word $w \in \Sigma^*$ such that $uw \in L \Leftrightarrow vw \notin L$. The observation table works in a similar way: Each representative $u \in R \cup R \cdot \Sigma$ represents an equivalence class while the samples from S are used to distinct them. We know that two representatives $u, v \in R \cup R \cdot \Sigma$ are not L-equivalent if there is a sample $w \in S$ that distinguishes their equivalence classes, i.e. $uw \in L \Leftrightarrow vw \notin L$. In this case, $T(uw) \neq T(vw)$ holds and the w-entry of the rows of u and v in the observation table are different. Since we want to compute a congruence, we also have to take care of the a-successors of representatives. Therefore, we need to store information about the words in R and $R \cdot \Sigma$.

On the other hand, we cannot tell generally speaking whether or not two representatives u, v are L-equivalent only using the information stored in the observation table. It may happen that there is no $w \in S$ that can be used to distinguish the equivalence classes of u and v but in fact $u \not\sim_L v$. In this case, both rows of the observation table are equal, constituting why an observation table only stores an approximation of the Nerode congruence. Let us formalize this intuition.

Definition 3.3. Let $O = (R, S, T)$ be an observation table for the target language L. Two words $u, v \in R$ are called *O-equivalent*, denoted by $u \sim_O v$, if and only if

$$T(uw) = T(vw)$$

holds for all $w \in S$.

$\triangleleft$

The O-equivalence defines a partition on the set R of representatives. We denote the equivalence class of a word $u \in R$ by $[\![u]\!]_O$ and the number of O-equivalence classes by $index(\sim_O)$. Since two O-equivalent representatives have the same row, one can think of a representative's row as a unique pattern for this equivalence class. Note that this definition of O-equivalence differs from the one given in Chapter 5 (cf. Definition 5.18) where we define it for all words over the given alphabet. Finally, we observe that two L-equivalent representatives u and v are clearly O-equivalent since there is no word which can be used to distinguish $[\![u]\!]_O$ and $[\![v]\!]_O$. As a direct consequence, it follows that $index(\sim_O) \leq index(\sim_L)$.

It can happen that the information stored in an observation table is incomplete in the sense that it defines an equivalence relation but no congruence relation. We can formulate two conditions, which guarantee that the information maintained by an observation table defines in fact a congruence relation.

- We call an observation table *closed* if and only if the condition

$$\forall u \in R \; \forall a \in \Sigma : \; [\![ua]\!]_O \cap R \neq \emptyset$$

holds. Intuitively, for each representative $r \in R$ and each $a \in \Sigma$ the row of the representative ua has to be existing in the observation table.

- We call an observation table *consistent* if and only if the condition

$$\forall u, v \in R \; \forall a \in \Sigma : \; u \sim_O v \Rightarrow ua \sim_O va$$

holds. This means that the equivalence $\sim_O$ is sound w.r.t. concatenation. Intuitively, every a-successors of two representatives that have the same row have to have the same row, too.

It is not hard to verify that $\sim_O$ is in fact a congruence if O is both closed and consistent. We can use $\sim_O$ to construct a DFA in the same manner as the Nerode congruence. The states of the DFA are the $\sim_O$-equivalence classes and the transition function is defined according to the congruence. The DFA $\mathcal{A}_O$ is used as conjecture.

Construction 3.4. *Let O be a closed and consistent observation table for the language L. We define the DFA $\mathcal{A}_O = (Q_O, \Sigma, q_0^O, \delta_O, F_O)$ as follows:*

- $Q_O = \{[\![u]\!]_O \mid u \in R\}$,
- $q_0^O = [\![\varepsilon]\!]_O$,
- $\delta_O([\![u]\!]_O, a) = [\![ua]\!]_O$,
- $F_O = \{[\![u]\!]_O \in Q_O \mid T(u \cdot \varepsilon) = 1\} = \{[\![u]\!]_O \in Q_O \mid u \in L\}$.

Each state of $\mathcal{A}_O$ corresponds to a unique row. The initial state is represented by the row of ε. Note that this definition is sound: The initial state is well defined since ε is always a representative and because ε is always a sample, too, the set of final states is also well defined. Moreover, the transition function δ_O is sound since $\sim_O$ is a congruence. Let us argue that that $\mathcal{A}_O$ works correctly on all representatives.

Lemma 3.5. *Let $O = (R, S, T)$ be a closed and consistent observation table for a regular language L. Then, for all $u \in R$ the condition*

$$u \in L(\mathcal{A}_O) \;\Leftrightarrow\; T(u) = 1 \;\Leftrightarrow\; u \in L$$

holds.

Proof of Lemma 3.5. To prove Lemma 3.5, we show that

$$[\![\varepsilon]\!]_O \xrightarrow{u}_{\mathcal{A}_O} [\![u]\!]_O$$

holds for all representatives $u \in R$ by a straightforward induction over the length of u.

Let $u = \varepsilon$. Then, the lemma holds by definition of runs of DFAs.

Let $u = va$. Since R is prefix closed, we can apply the induction hypothesis and gain

$$[\![\varepsilon]\!]_O \xrightarrow{v}_{\mathcal{A}_O} [\![v]\!]_O \,.$$

Input: A MAT for a regular language $L \subseteq \Sigma^*$
Output: The DFA $\mathcal{A}_L$

$O = (R, S, T)$ with $R = \{\varepsilon\}$, $S = \{\varepsilon\}$ and $T(w) = MAT(w)$ for all $w \in \{\varepsilon\} \cup \Sigma$;

repeat
  while *O is not closed or not consistent* **do**
    if *O is not consistent* **then**
      Choose $u, v \in R$ with $u \sim_O v$ and $w \in S$, $a \in \Sigma$ such that $T(uaw) \neq T(vaw)$;
      $S := S \cup \{aw\}$;
      update(O) ;
    end
    if *O is not closed* **then**
      Choose a $u \in R$ and an $a \in \Sigma$ such that $[\![ua]\!]_O \cap R = \emptyset$;
      $R := R \cup \{ua\}$;
      update(O) ;
    end
  end
  Construct the conjecture A_O ;
  if *the MAT provides a counter-example w* **then**
    $R := R \cup pref(w)$;
    update(O) ;
  end
until $L(\mathcal{A}_O) = L$;
return $\mathcal{A}_O$;

Algorithm 2: Angluin's learner

Moreover, we know from the definition of δ_O that $\delta_O([\![v]\!]_O) = [\![va]\!]_O$ holds and, hence,

$$[\![\varepsilon]\!]_O \xrightarrow{v}_{\mathcal{A}_O} [\![v]\!]_O \xrightarrow{a}_{\mathcal{A}_O} \underbrace{[\![va]\!]_O}_{[\![u]\!]_O} .$$

To prove Lemma 3.5, consider a word $u \in R$. Since $F = \{[\![u]\!]_O \mid u \in L\}$ and $[\![\varepsilon]\!]_O \xrightarrow{u}_{\mathcal{A}_O} [\![u]\!]_O$, we gain

$$u \in L(\mathcal{A}_O) \;\Leftrightarrow\; [\![\varepsilon]\!]_O \xrightarrow{u}_{\mathcal{A}_O} [\![u]\!]_O \in F \;\Leftrightarrow\; T(u) = 1 \;\Leftrightarrow\; u \in L .$$

□

Let us now have a closer look at Angluin's learner as it is presented in Algorithm 2. The algorithm starts with an initial observation table $O = (R, S, T)$ with $R = \{\varepsilon\}$ and $S = \{\varepsilon\}$. Furthermore, the mapping T contains the values of each word u from $\{\varepsilon\} \cup \Sigma$. These values are obtained by asking membership queries.

During the run of the algorithm, it can happen that the observation table is not closed or not consistent or neither closed nor consistent. Before we can construct a conjecture, we have to ensure that the observation table is closed and consistent.

If O is not consistent, then there are O-equivalent representatives $u \sim_O v \in R$ and $a \in \Sigma$ such that $ua \not\sim_O va$. That means that there is a sample $w \in S$ with $T(uaw) \neq T(vaw)$. If we add aw to S, then the rows of u and v become different (the consistent condition is then satisfied for u and v) and the number of $\sim_O$ equivalence classes increases. Since a new column is added to the table, we may have to update it by by asking membership questions for all new table entries. This is done by the function update(O).

If O is not closed, we know that there is a $u \in R$ and an $a \in \Sigma$ such that $[\![ua]\!]_O \cap R = \emptyset$. That means that there is no representative $v \in R$, which has the same row as ua. In this case, we add ua to R and update O. Thereby, the observation table gains a new row (the closed condition is then satisfied for u and a) and the number of $\sim_O$ equivalence classes increases. We repeat both operations until O is closed and consistent. Note that both operations preserve the prefix-closedness of R and the suffix-closedness of S.

If O is a closed and consistent observation table, we construct a conjecture $\mathcal{A}_O$ and ask the teacher whether $\mathcal{A}_O$ is equivalent to L. In the case that $\mathcal{A}_O$ is not equivalent to L, we have not gathered enough information about L yet and the teacher replies with a counter-example w. We then add w and all of its prefixes $u \in pref(w)$ to R and update the table. This also preserves the prefix-closedness of R.

We repeat this until the conjecture $\mathcal{A}_O$ is equivalent to $\mathcal{A}_L$. This directly shows that the algorithm returns an equivalent DFA if it terminates. Therefore, it remains to show that the algorithm eventually terminates. Before we do so, let us consider the following example.

Example 3.6. Consider the language L defined by the automaton $\mathcal{A}_L$ in Figure 3.1(d). The initial observation table O_1 is depicted in Figure 3.1(a). This observation table is not closed since $[\![b]\!]_{O_1} \cap R_1 = \emptyset$, i.e. there is no representative v such that the row of b and the row of v coincide. Therefore, the word b is added to R_1 and the observation table is updated by asking membership queries for ba and bb.

This yields the observation table O_2 depicted in Figure 3.1(b). O_2 is both closed and consistent and we construct the conjecture $\mathcal{A}_{O_2}$ which is also shown in the same figure. Since $\mathcal{A}_{O_2}$ is not equivalent to $\mathcal{A}_L$, the teacher provides the counter-example aa as reply to the equivalence query. The word aa is then added to R_2 and the observation table is updated. This yields the observation table O_3 depicted in Figure 3.1(c).

Now, consider the representatives $\varepsilon \sim_{O_3} a \in R$ and $a \in \Sigma$. Since $a \not\sim_{O_3} aa$, the observation table O_3 is not consistent. The word a is added to S and T_3 is updated. Note that the rows of a and aa have become different.

The observation table O_4 is again closed and consistent and the learner asks an equivalence query on $\mathcal{A}_{O_4}$. Since the conjecture is equivalent to $\mathcal{A}_L$, the algorithm terminates.

$\triangleleft$

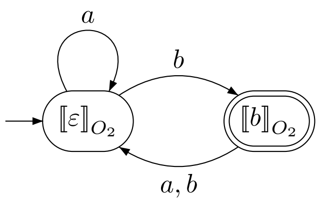
Let us now turn on proving Theorem 3.1.

Proof of Theorem 3.1. As already mentioned, we observe that $index(\sim_O) \leq index(\sim_L)$. If O is closed and consistent, then $index(\sim_O) = |Q_O|$ where Q_O is the state set of $\mathcal{A}_O$. Furthermore, we know that after termination $L(\mathcal{A}_O) = L$ holds. Therefore, $\mathcal{A}_O$ has to be the minimal DFA for L.

O_1	ε
ε	0
a	0
b	1

(a) The observation table O_1

O_2	ε
ε	0
b	1
a	0
ba	0
bb	0

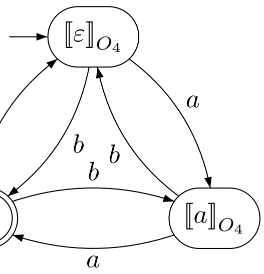


(b) The observation table O_2 and the conjecture $\mathcal{A}_{O_2}$

O_3	ε
ε	0
a	0
b	1
aa	1
ab	0
ba	0
bb	0
aaa	0
aab	0

(c) The observation table O_3

O_4	ε	a
ε	0	0
a	0	1
b	1	0
aa	1	0
ab	0	0
ba	0	0
bb	0	1
aaa	0	0
aab	0	1



(d) The observation table O_4 and the conjecture $\mathcal{A}_{O_4} = \mathcal{A}_L$

Figure 3.1: The observation tables and the conjectures of Example 3.6

It remains to show that the algorithm terminates eventually. Thereto, we make two observations.

1. After finitely many steps the observation table is closed and consistent.

   Assume that O is not closed. Then, a new representative, whose equivalence class is not yet present in the set R, is added to R and $index(\sim_O)$ is increased.

   Now, assume that O is not consistent. Then, there are two representatives $u \sim_O v \in R$ and $a \in \Sigma$ such that $ua \not\sim_O va$. The algorithm adds the sample aw to S, which yields $T(uaw) = 1 \Leftrightarrow T(vaw) = 0$. Therefore, in the new observation table u and v are no longer O-equivalent, i.e. $index(\sim_O)$ is increased.

   Since $index(\sim_O) \leq index(\sim_L)$, it can only happen finitely many times that an observation table is not closed or consistent.

2. The number of states of a conjecture is strictly smaller than the number of state of the succeeding conjecture.

   Let $\mathcal{A}_O$ be a conjecture and $\mathcal{A}_{O'}$ the succeeding conjecture. Furthermore, let w be the counter-example for $\mathcal{A}_O$. As all prefixes of w are added to R, we assume that $index(\sim_O)$ is not increased. Then, the observation table is no longer closed and consistent. Otherwise, we would gain the same conjecture as before, i.e. $\mathcal{A}_O = \mathcal{A}_{O'}$. Additionally, we know from Lemma 3.5 that $\mathcal{A}_{O'}$ works correctly on all words of R'. Since $\mathcal{A}_O = \mathcal{A}_{O'}$, we deduce that $\mathcal{A}_O$ also works correctly on all words of R' and in particular on w. This is a contradiction since w is a counter-example.

   Therefore, $index(\sim_O)$ is increased every time a counter-example is provided. Again, this can happen only finitely many times.

Eventually, $index(\sim_O) = index(\sim_L)$ since $index(\sim_L)$ is finite for every regular language L. Thus, the algorithm terminates eventually.

It is left to have a look at the runtime complexity of the algorithm. Let n be the number of states of $\mathcal{A}_L$ and m the length of the longest counter-example presented by the teacher. From the observations one and two above we can deduce:

- The observation table is not closed at most $n-1$ times. By adding a new representative to R, we have to make at most $|\Sigma| \cdot |S|$ membership queries.

- The observation table is not consistent at most $n-1$ times. By adding a new sample to S, we have to make at most $|\Sigma| \cdot |R|$ membership queries.

- There can be at most $n-1$ wrong conjectures. By adding a counter-example, R may be increased by at most m. Therefore, we have to make at most $m \cdot |\Sigma| \cdot |S|$ membership queries.

Overall, we can make at most n equivalence queries and $\mathcal{O}(mn^2)$ membership queries which yields a polynomial runtime of the algorithm.

□

3.2 Regular positive, negative inference

Gracía and Oncina [GO92] proposed an algorithms for inferring a DFA from positive and negative samples of a given regular language L. In contrast to Angluin's learner of the previous section, this method is a non-query algorithm and does not rely on a teacher. Instead, it is provided with a finite *sample* $S = (S_+, S_-)$ to infer a DFA for the language L. Thereby, a sample S consists of a nonempty set of *positive samples* $S_+ \subseteq L$ and a set of *negative samples* S_- with $S_- \cap L = \emptyset$. The inference results in a DFA, which is compatible with S in the sense that it accepts all positives samples from S_+ and rejects all negative samples from S_-. In other words the construction yields a DFA $\mathcal{A}_S$ with $S_+ \subseteq L(\mathcal{A}_S)$ and $L(\mathcal{A}_S) \cap S_- = \emptyset$.

However, we cannot guarantee in general that the inference results in a DFA with $L(\mathcal{A}_S) = L$ since the sample S may not contain enough information about the language L. Therefore, we show the existence of *complete samples* for every regular language. These complete samples fulfill the following properties.

- If S is a complete sample, then $L(\mathcal{A}_S) = L$.
- For every regular language L the size of a complete sample is polynomial in the size of the minimal DFA for L.
- If (S_+, S_-) is a complete sample and $S_+ \subseteq S'_+$ and $S_- \subseteq S'_-$, then the sample $S' = (S'_+, S'_-)$ is also complete. We call the sample (S'_+, S'_-) an *extension* of S.

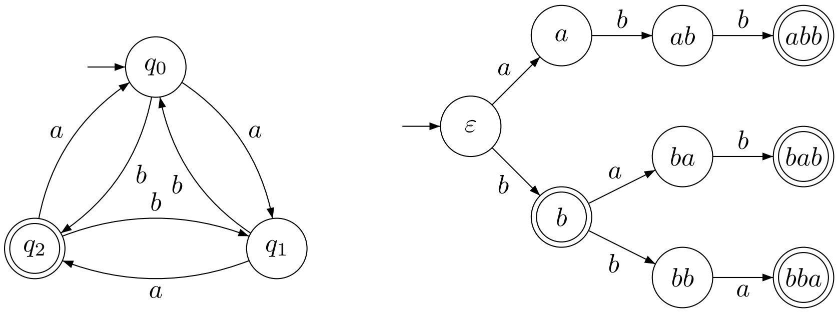
We show that $\mathcal{A}_S$ can be constructed in time polynomial in the size of S. To be precise, Gracía and Oncina's construction is polynomial in the sum of the length of all words in the sample S. Furthermore, we show that the *regular positive, negative inference algorithm (RPNI)* constructs the minimal DFA for L if provided with a complete sample for L.

The idea of the construction is to start with the DFA that recognizes exactly the words in S_+. Note that this DFA is obviously compatible with the given sample S. Then, we try to merge states of this DFA to gain a DFA that maybe accepts more words but is still compatible with S. Since there are exponentially many possibilities to merge states of a DFA, we cannot use an exhaustive search to discover a "good" selection of states. Therefore, we do the state merging in a certain order. If the merged automaton is not compatible with the sample, we discard it. Otherwise, we proceed with this new DFA and try to merge more states.

This procedure results in a sequence $\mathcal{A}_0, \ldots, \mathcal{A}_n$ of DFAs. Since merging states of finite automata can be seen as building equivalences over the set of their states and constructing the quotient automaton afterwards, we know from Section 2.1 that $L(\mathcal{A}_0) \subseteq \ldots \subseteq L(\mathcal{A}_n)$. Furthermore, we show that, if S is a complete sample, then $L(\mathcal{A}_n) = L$ holds.

Before we describe Gracía's and Oncina's construction in detail, we need to fix some notations. For this section, let Σ be a finite alphabet, $L \subseteq \Sigma^*$ a regular language and $S = (S_+, S_-)$ be a sample with $S_+ \subseteq L$ and $S_- \cap L = \emptyset$.

For a sample $S = (S_+, S_-)$ we define a DFA $\mathcal{A}(S_+)$, which recognizes exactly the words of S_+. The idea is to use all prefixes of the words of S_+ as states of the DFA.



(a) The DFA $\mathcal{A}$ recognizing $L = \{w \in \Sigma^* \mid (|w|_b - |w|_a) \mod 3 = 1\}$

(b) The automaton $\mathcal{A}(S_+)$

Figure 3.2: The automata of Example 3.8

Construction 3.7. *Let $S = (S_+, S_-)$ be a sample. Then, we define the DFA $\mathcal{A}(S_+) = (Q_+, \Sigma, q_0^+, \delta_+, F_+)$ as follows:*

- $Q_+ = pref(S_+)$,
- $q_0^+ = \varepsilon$,
- $F_+ = S_+$ *and*
- δ_+ *defined by*

$$\delta_+(w,a) = \begin{cases} wa & \text{, if } wa \in Q_+ \\ \text{undefined} & \text{, else} \end{cases} .$$

Note that for each sample the set S_+ has to be nonempty. Thus, the DFA $\mathcal{A}(S_+)$ contains at least the state ε.

The definition of δ_+ leaves some transitions undefined and, hence, $\mathcal{A}(S_+)$ is a *partial DFA*. We can fix this by adding a sink state to Q_+ and let every non-defined transition point to this new state. It turns out that it is easier for the following considerations to work with a partial DFA. However, we do not use this property explicitly but keep this in mind when it comes to the construction of equivalence classes over the states of $\mathcal{A}(S_+)$ as well as quotient automata.

Finally, we call an equivalence relation $\sim$ over Q_+ a *congruence* if for all $u, v \in Q_+$ with $u \sim v$ and $a \in \Sigma$ the condition

$$ua, va \in Q_+ \;\Rightarrow\; ua \sim va$$

holds. Note that this definition of a congruence has the additional requirement $ua, va \in Q_+$ to respect the undefined transitions of $\mathcal{A}(S_+)$.

Example 3.8. We reconsider the language $L = \{w \in \Sigma^* \mid (|w|_b - |w|_a) \mod 3 = 1\}$ from Example 3.6 in Section 3.1. A DFA $\mathcal{A}$ recognizing L is depicted in Figure 3.2(a).

In this example we consider the sample $S = (S_+, S_-)$ with

$$S_+ = \{bab, bba, abb\} \text{ and } S_- = \{\varepsilon, a, ab, ba\} .$$

The automaton $\mathcal{A}(S_+)$ is depicted in Figure 3.2(b).

◁

Input: A sample $S = (S_+, S_-)$
Output: The DFA $\mathcal{A}_S$

Construct the automaton $\mathcal{A}(S_+)$;
Order the set $Q_+ = \{u_0, \ldots, u_n\}$ in canonical order;
Set $\sim_0 = \{(u, u) \mid u \in Q_+\}$;
for $i = 1, \ldots, n$ **do**
    if $u_i \not\sim_{i-1} u_j$ *for all* $j \in \{0, \ldots, i-1\}$ **then**
        $j := 0$;
        repeat
            Let $\sim$ be the smallest congruence that contains $\sim_{i-1}$ and the pair (u_i, u_j);
            $\mathcal{B} = \mathcal{A}(S_+)_{/\sim}$;
            $j := j + 1$;
        until $L(\mathcal{B}) \cap S_- = \emptyset$;
        $\sim_i = \sim$;
    else
        $\sim_i = \sim_{i-1}$;
    end
end
$\mathcal{A}_S = \mathcal{A}(S_+)_{/\sim_n}$;
return $\mathcal{A}_S$;

Algorithm 3: The RPNI algorithm

As mentioned above, we merge the states of $\mathcal{A}(S_+)$ in a certain order to ensure polynomial runtime of the algorithm. Since the states of $\mathcal{A}(S_+)$ are words over Σ, we use the canonical order $<$ to order the state set Q_+. Then, we process every state in ascending order starting with the smallest state $u \neq \varepsilon$ as follows.

We try to merge the current state with all smaller states successively in ascending order until one of the following situations occurs: Either we discover a DFA that is compatible with S or there are no more smaller states. In the first case, we proceed with the merged DFA while we do not merge the current state in the latter case. However, in both cases, we continue with the next bigger state until we processed every state of Q_+.

To make sure that a mergence results in a deterministic finite automaton we may have to merge additional states cascadingly. Formally, we do a mergence by defining a congruence relation $\sim$ over the set of states. This congruence relation groups all merged states so far in their equivalence classes respectively. Afterwards, we construct the quotient automaton $\mathcal{A}(S_+)_{/\sim}$.

Because of the cascading mergence, it can happen that a currently processed state is already merged with a smaller state. Then, we simply skip this state and proceed with the next one.

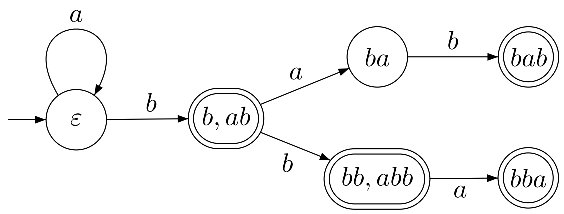
The RPNI algorithm in pseudo code can be found in Algorithm 3. It constructs a DFA $\mathcal{A}_S$ from a given sample $S = (S_+, S_-)$ in polynomial time. As we argue later, $\mathcal{A}_S = \mathcal{A}_L$ holds if the algorithm is provided with a complete sample for the language L.

Example 3.9. We continue Example 3.8 and describe a run of the RPNI algorithm on the sets

$$S_+ = \{bab, bba, abb\} \text{ and } S_- = \{\varepsilon, a, ab, ba\} \, .$$

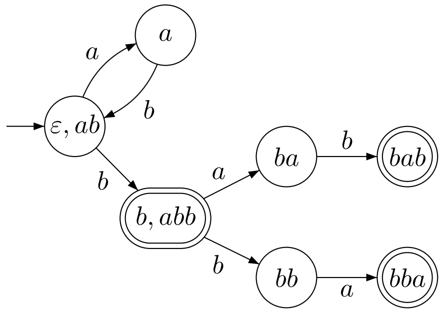
The algorithm starts with the automaton $\mathcal{A}(S_+) = (Q_+, \Sigma, q_0^+, \delta_+, F_+)$ depicted in Figure 3.2(b) and orders the set $Q_+ = \{\varepsilon, a, b, ab, ba, bb, abb, bab, bba\}$ w.r.t. the canonical order. In the following, let i denote the loop index of the RPNI algorithm.

Let $i = 1$ and $u_i = a$. The automaton $\mathcal{A}(S_+)_{/\sim_1}$ where $\varepsilon \sim_1 a$ is shown below. Since $\mathcal{A}(S_+)_{/\sim_1}$ accepts $ab \in S_-$, the condition $\mathcal{A}(S_+)_{/\sim_1} \cap S_- \neq \emptyset$ is violated. Therefore, $\sim_1$ is no valid congruence and the automaton is discarded. Thus, $\sim_1 = \sim_0$ and since there is no smaller state to merge a with, we proceed with the old automaton $\mathcal{A}(S_+)$.

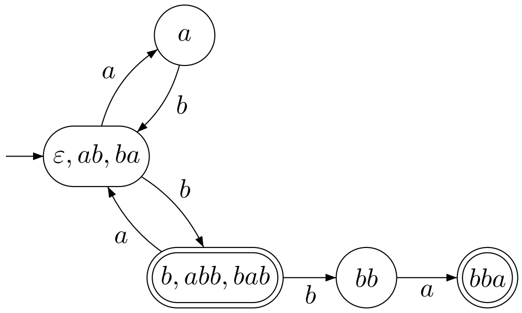


Let $i = 2$ and $u_2 = b$. As above, u_2 cannot be merged with ε or a. Hence, $\sim_2 = \sim_0$.

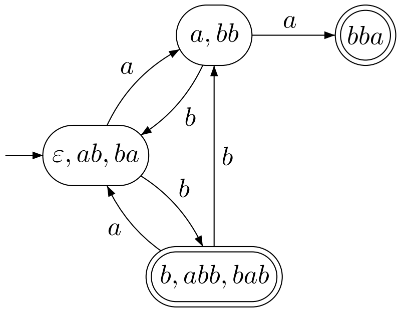
Let $i = 3$ and $u_3 = ab$. The congruence $\sim_3$ with $\varepsilon \sim_3 ab$ is valid since $\mathcal{A}(S_+)_{/\sim_3}$ does not accept any word from S_-. The automaton $\mathcal{A}(S_+)_{/\sim_3}$ is shown below.



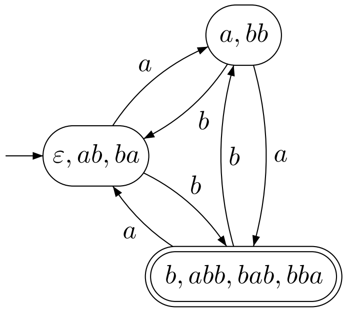
Let $i = 4$ and $u_4 = ba$. The congruence $\sim_4$ with $\varepsilon \sim_4 ba$ is also valid. The automaton $\mathcal{A}(S_+)_{/\sim_4}$ is depicted below.



Let $i = 5$ and $u_5 = bb$. The congruence $\sim_5$ with $\varepsilon \sim_5 bb$ is not valid. However, the congruence $\sim_5$ with $a \sim_5 bb$ is valid. The automaton $\mathcal{A}(S_+)_{/\sim_5}$ is shown below.



For $i = 6$ and $i = 7$ the words $u_6 = abb$ and $u_7 = bab$ are already $\sim_{i-1}$-equivalent to b. Thus, the algorithm sets $\sim_5 = \sim_6 = \sim_7$ and proceeds with $i = 8$ and $u_8 = bba$. Finally, the automaton $\mathcal{A}(S_+)_{/\sim_8} = \mathcal{A}_S$ is gained. It is depicted below. This automaton is also the minimal automaton recognizing L.



◁

Let us now have a closer look at the runtime complexity of the RPNI algorithm.

Lemma 3.10. *Let $S = (S_+, S_-)$ be a sample and $n = |Q_+|$ the number of states of the automaton $\mathcal{A}(S_+)$. Moreover, let m be the sum of the length of all words in S_-, i.e. $m = \sum_{w \in S_-} |w|$. The runtime of the RPNI algorithm is bounded by $\mathcal{O}(n^4 \cdot m)$.*

Proof of Lemma 3.10. We first state that n is polynomial in the size of the sample S. To be precise, $n \leq 1 + \sum_{w \in S_+} |w|$. Since the states of the automaton $\mathcal{A}(S_+)$ are only merged with canonically smaller states, there are at most

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$$

merging operations. Each merging operation can be done in time $\mathcal{O}(n \cdot |\Sigma|)$ since we may have to update the transitions of every state. Recall that a merging operation can cause more states to be merged to gain a deterministic finite automaton. Since each of these cascading mergence decreases the number of states by one, this can only happen n times.

After each merging operation, we have to check whether the resulting DFA is still compatible with the sample. Thereto, we simulate the run of the merged automaton on every sample of S_-. Only if all negative samples are rejected, we proceed. This simulation can be done in $\mathcal{O}(m)$.

All in all the runtime complexity of the RPNI algorithm is $\mathcal{O}(n^4 \cdot m \cdot |\Sigma|)$ where $|\Sigma|$ is a constant. Thus, the RPNI algorithm is polynomial in the sum of the length of all samples in S.

$\square$

We now turn on the question how to guarantee the inference of the minimal DFA recognizing L. It is immediately clear that a sample has to contain enough information about the language in order to infer the minimal DFA for L. However, it is not obvious how to formulate this requirement.

Gracía and Oncina's characterization of "enough information" is based on a language theoretic description of the DFA $\mathcal{A}_L$. They define sets of words $MR(L)$ and $MTR(L)$, which code the DFA $\mathcal{A}_L$. To be precise, the set $MR(L)$ contains the minimal representatives of each L-equivalence class and, thus, a representative of each state of $\mathcal{A}_L$. Moreover, the set $MTR(L)$ contains a word for each transition in $\mathcal{A}_L$.

Since the automaton $\mathcal{A}(S_+)$, and finally $\mathcal{A}(S_+)_{/\sim_n}$, is constructed from the prefixes of words in S_+, we have to ensure that the sample S already contains representatives of all states and transitions of $\mathcal{A}_L$. Moreover, we have to make sure that the right states are marked as final and that there are words in S_- that prevent the RPNI algorithm from merging non L-equivalent states.

The following definition uses the sets $MR(L)$ and $MTR(L)$ to formalize these intuitive requirements.

Definition 3.11. [Complete sample] Let $L \subseteq \Sigma^*$ be a regular language over the alphabet Σ. The set

$$MR(L) = \{w \in \Sigma^* \mid w \in \mathit{pref}(L) \text{ and } \forall u \sim_L w : \; w < u\}$$

is called the set of *minimal representatives* of L. The set

$$MTR(L) = \{w \in \Sigma^* \mid w \in \mathit{pref}(L) \text{ and } w = ua \text{ for a } u \in MR(L), a \in \Sigma\}$$

is called the set of *minimal transition representatives* of L.

A sample $S = (S_+, S_-)$ is said to be *complete* for a regular language L if the following conditions hold:

1. For all $w \in L$ there is a $u \in S_+$ with $u \sim_L w$.
2. For all $w \in MTR(L)$ there is a $v \in \Sigma^*$ such that $wv \in S_+$.
3. For all $u \in MR(L)$ and for all $v \in MTR(L)$ with $u \not\sim_L v$ there is a $w \in \Sigma^*$ such that

$$uw, vw \in S_+ \cup S_- \text{ and } uw \in S_+ \Leftrightarrow vw \in S_- .$$

$\triangleleft$

Intuitively, condition 1. ensures that every final state of the automaton $\mathcal{A}_L$ has a representative in S_+ and hence in $\mathcal{A}(S_+)$. Furthermore, the second condition requires every transition of $\mathcal{A}_L$ to be present in $\mathcal{A}(S_+)$. Finally, condition 3. ensures that the RPNI algorithm has enough information to distinguish all L-equivalence classes. This condition is needed to make sure that every L-equivalence class is represented by at least one state of $\mathcal{A}(S_+)$. Moreover, it prevents the mergence of non L-equivalent states.

Example 3.12. We continue Example 3.8. The sets $MR(L)$ and $MTR(L)$ are given by

$$MR(L) = \{\varepsilon, a, b\} \text{ and } MTR(L) = \{a, b, aa, ab, ba, bb\} .$$

In Figure 3.2(a), the minimal representatives of L coincide with the states of $\mathcal{A}$. To be precise, ε coincides with state q_0 while a coincides with state q_1 and b with q_2. Additionally, all transitions of $\mathcal{A}$ are covered by the set $MTR(L)$.

$\triangleleft$

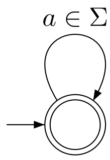
Note that the sample of Example 3.8 is not complete since $\varepsilon \in MR(L)$ and $a \in MTR(L)$ but there is no $w \in \Sigma^*$ such that $w, aw \in S_+ \cup S_-$. However, as shown in Example 3.9 the automaton A_S does recognize the language L. Moreover, the DFA is the minimal DFA for the language L. This shows that the RPNI algorithm may also work well on samples that are not complete.

The next example shows that this does not hold in general.

Example 3.13. Let Σ be an alphabet, $L \neq \Sigma^*$ a regular language and $S = (S_+, S_-)$ a sample such that the following properties are fulfilled.

- For every symbol $a \in \Sigma$ there is a words $w = uav \in S_+$ with $u, v \in \Sigma^*$.
- $S_- = \emptyset$.

Since there is no negative sample, which prevents the RPNI algorithm from merging two arbitrary states, the algorithm constructs the DFA $\mathcal{A}_S$ shown below.



Clearly, this DFA does not recognize L because $L(\mathcal{A}) = \Sigma^*$.

$\triangleleft$

However, the following theorem shows that the RPNI algorithm always yields the minimal DFA for a given regular language if provided with a complete sample.

Theorem 3.14 (Gracía and Oncina [GO92], Löding)**.** *If $S = (S_+, S_-)$ is a complete sample for a regular language L, then the RPNI algorithm yields the minimal DFA for L.*

To prove this theorem, we first show the following lemma.

Lemma 3.15. *Let S be a complete sample for the regular language L and i the loop index of a run of the RPNI algorithm. Then, for all $i \in \{0, \ldots, n\}$ holds:*

(a) $\forall u, v \in Q_+ : \; u \sim_i v \Rightarrow u \sim_L v$

(b) $\forall u, v \in \{u_0, \ldots, u_i\} : \; u \sim_L v \Rightarrow u \sim_i v$

Intuitively, part (a) of Lemma 3.15 states that the RPNI algorithm does never merge wrong states if provided with a complete sample. Moreover, part (b) states that two states are merged if this is necessary to gain the minimal DFA for L.

The proof of Lemma 3.15 is technical but may reveal some interesting details about the function of the RPNI algorithm.

Proof of Lemma 3.15. We show Lemma 3.15 by induction over i.

Let $i = 0$. Then, (a) and (b) are true because $\sim_0 = \{(u, u) \mid u \in Q_+\}$.

Let $i > 0$. As in the RPNI algorithm, we distinguish two cases:

1. There is a $j < i$ such that $u_i \sim_{i-1} u_j$. Then, the RPNI algorithm sets $\sim_i = \sim_{i-1}$.
   (a) Part (a) follows directly from the induction hypothesis.
   (b) Let $u_i \sim_L v$ for a $v \in \{u_0, \ldots, u_{i-1}\}$. Then, we can deduce that $u_j \sim_L v$ as follows:
   Since $u_i \sim_{i-1} u_j$, we can apply the induction hypothesis (a) and gain $u_i \sim_L u_j$. This yields
   $$u_i \sim_L v \sim_L u_j \; .$$
   Because $v, u_j \in \{u_0, \ldots, u_{i-1}\}$, we can apply the induction hypothesis (b) and gain $u_j \sim_{i-1} v$. Overall, we conclude that
   $$u_i \sim_{i-1} u_j \sim_{i-1} v$$
   and, thus, $u_i \sim_i v$ since $\sim_{i-1} = \sim_i$.

2. $u_i \not\sim_{i-1} u_j$ holds for all $j \in \{0, \ldots, i-1\}$.

   Preliminary, we show that $u_i \in MTR(L)$. Let $u_i = ua$. By definition of $MTR(L)$, we have to show that $u \in MR(L)$. Thereto, let $u' \in MR(L)$ such that $u' \sim_L u$ and assume $u' \neq u$. Then, we know $u' < u$ and thus $u' < u < ua$. Furthermore, $u'a \in Q_+$ because of part 2. of the definition of complete samples and, thus, $u'a \in MTR(L)$. Since $u, u' \in Q_+$ we can apply the induction hypothesis (b) and gain
   $$u' \sim_L u \;\Rightarrow\; u' \sim_{i-1} u \; .$$

Because $\sim_{i-1}$ is a congruence, $u'a \sim_{i-1} ua$ also holds.

This is a contradiction because $u'a < ua$ (since $u' < u$) and thus $u_i = ua \sim_{i-1} u'a$ but $u_i \not\sim_{i-1} u_j$ for all $j < i$ by assumption. Hence, $u_i \in MTR(L)$.

(a) To prove part (a) we show the following: If $v \in \{u_o, \ldots, u_{i-1}\}$, then u_i and v are not merged. In other words the smallest congruence $\sim$ that contains $\sim_{i-1}$ and (u_i, v) violates the condition $L(\mathcal{A}(S_+)_{/\sim}) \cap S_- = \emptyset$. Let $w \in MR(L)$, and thus, $w \in Q_+$, such that $w \sim_L v$. By applying the induction hypothesis (b) we gain $w \sim_{i-1} v$. Condition 3. of complete samples guarantees the existence of a w' such that

$$ww', u_i w \in S_+ \cup S_- \quad \text{and} \quad ww' \in S_+ \Leftrightarrow u_i w' \in S_-$$

since $u_i \in MTR(L)$.
Now, assume that v and u_j are merged, i.e. let $\sim$ be the smallest congruence that contains $\sim_{i-1}$ and (u_i, v). Then, we know that $v \sim w$ holds since $v \sim_{i-1} w$ and, hence, $w \sim u_i$. Thus, both words ww' and $u_i w'$ are accepted by $\mathcal{A}(S_+)_{/\sim}$. This violates the condition $L(\mathcal{A}(S_+)_{/\sim}) \cap S_- = \emptyset$. Therefore, u_i and v are not merged.

(b) Assume $u_i \sim_L u_j$ for a $j < i$ and let $\sim$ be the smallest congruence that contains $\sim_{i-1}$ and (u_i, u_j). We define a mapping

$$h: (Q_+)_{/\sim} \to (\Sigma^*)_{/\sim_L} \quad \text{with} \quad h([\![u]\!]_\sim) = [\![u]\!]_{\sim_L}$$

that maps every state of $\mathcal{A}(S_+)_{/\sim}$ to a state of $\mathcal{A}_L$. From condition (a), we know that this mapping is well defined. Moreover, it is not difficult to see that h respects the transitions and final states and, therefore, is a homomorphism. Thus, $L(\mathcal{A}(S_+)_{/\sim}) \subseteq L(\mathcal{A}_L) = L$ holds and, hence, $L(\mathcal{A}(S_+)_{/\sim}) \cap S_- = \emptyset$, too.
This means that the algorithm merges u_i and u_j and, hence, $u_i \sim_i u_j$.

□

The proof of Lemma 3.15 shows the importance of the order in which the states of $\mathcal{A}(S_+)$ are merged. However, the inference of the DFA $\mathcal{A}_L$ can only be guaranteed for complete samples.

We are now able to prove Theorem 3.14 by using the results of Lemma 3.15.

Proof of Theorem 3.14. To proof Theorem 3.14, we have show that the DFA $\mathcal{A}(S_+)_{/\sim_n}$ is isomorphic to the canonical DFA $\mathcal{A}_L$.

From Lemma 3.15 we know that

$$u \sim_L v \;\Leftrightarrow\; u \sim_n v$$

holds for all $u, v \in Q_+$. Thus, we can define a mapping

$$h: (Q_+)_{/\sim_n} \to (\Sigma^*)_{/\sim_L} \quad \text{with} \quad h([\![u]\!]_{\sim_n}) = [\![u]\!]_L \;.$$

In other words, h maps each state of the automaton $\mathcal{A}(S_+)_{/\sim_n}$ to a state of the canonical DFA $\mathcal{A}_L$. Since $u \sim_L v \Leftrightarrow u \sim_n v$, we know that the mapping h is well defined and injective. Moreover, condition 3. of complete samples ensures that there is a representative for every L-equivalence class in Q_+. Thus, h is also surjective and, therefore, bijective.

Finally, conditions 1. and 2. of complete samples ensure that the mapping h respects the final states and transitions of $\mathcal{A}_L$. Thus, h is an isomorphism from $\mathcal{A}(S_+)_{/\sim_n}$ to $\mathcal{A}_L$. Therefore, $\mathcal{A}(S_+)_{/\sim_n}$ is the minimal DFA recognizing L (up to isomorphism).

$\square$

As a final remark, we state that there is a complete sample for every regular language L that is polynomial in the size of the DFA $\mathcal{A}_L$:

- We know that $|MR(L)| = index(\sim_L)$ and thus $|MR(L)|$ equals the number of states of $\mathcal{A}_L$.
- Moreover, $|MTR(L)|$ equals the number of transitions of $\mathcal{A}_L$.
- Finally, for every $u \in MTR(L)$ and every $v \in MR(L)$ there is a $w \in \Sigma^*$ with $uw \in L \Leftrightarrow vw \notin L$ such that $|w| \leq index(\sim_L)$.

Hence, there is a complete sample for every regular language that is polynomial in the size of $\mathcal{A}_L$. Moreover, we observe that every extension of a complete sample is again a complete sample.

4 Validating XML word representations

In this chapter, we address the task of validating XML word representations (cf. Definition 2.11) under a constant memory assumption as it is studied by Segoufin and Vianu [SV02]. By validation, we mean the check whether an XML word representation is valid w.r.t. a given DTD in a single pass using only a fixed amount of memory. The amount of memory may depend on the DTD but not on the XML word representation processed.

A natural class of language acceptors that preserves these constraints is the class of (deterministic) finite automata. DFAs have no access to auxiliary memory and can only use their states to store information. The main objective of this section is to show how DFAs capable of validating XML word representation can be learned.

We concentrate on a variant of the validation task that Segoufin and Vianu simply call *validation*. This validation does not include the check for well formedness of the input since it is well known that this kind of well formedness is a context free property and, hence, DFAs are incapable of performing this check in general. Thus, we assume that the input of a DFA always is a valid XML word representation and require that the automaton works correctly on these well-formed words. On not well-formed words, i.e. on words that are no XML word representation, we do not care what the answer is. We think that this is not too restrictive in practice because most XML documents are generated automatically.

We can formalize this intuitive explanation as follows. Let L^{XML} be the language of all XML word representations of tree documents over Σ_{tag} and $d = (\Sigma_{tag}, r, \rho)$ a DTD. Then, *d can be validated* by a DFA $\mathcal{A}$ if and only if

$$L(d) = L^{XML} \cap L(\mathcal{A}) \ .$$

Such DTDs are called *recognizable* and we say that $\mathcal{A}$ *is equivalent to d*. However, note that that the DFA $\mathcal{A}$ is not unique for a given DTD since the behavior on non XML word representations can be arbitrary. To support the understanding, let us consider the following example.

Example 4.1. Consider the DTD $d = \begin{array}{c} r \to a \\ a \to a^? \end{array}$, which defines a strand of arbitrary length of nodes labeled by a and a single root node r. This DTD is recognizable by a finite automaton, whose accepted language is defined by the regular expression $ra^+\overline{a}^+\overline{r}$. Under the assumption that the input is well formed, a finite automaton only needs to check whether the input starts with r followed by a's and ends with $\overline{a}$'s followed by a single $\overline{r}$.

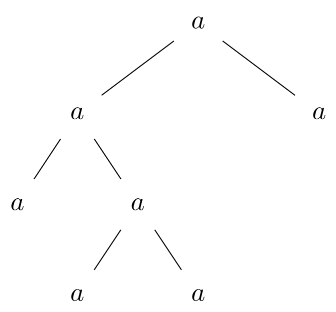


Figure 4.1: A tree document valid w.r.t. the DTD from Example 4.2

◁

Not every DTD is recognizable by a finite automaton. In fact, Segoufin and Vianu identified a necessary condition for a DTD to be recognizable. A non-recognizable DTD is presented in the following example.

Example 4.2. Consider the DTD $d = a \rightarrow aa \mid \varepsilon$. A Σ_{tag}-labeled tree, which is valid w.r.t. d, is depicted in Figure 4.1.

A finite automaton cannot validate this DTD. Intuitively, even if the input is assumed to be an XML word representation, the automaton cannot store enough information to decide whether a or $\overline{a}$ is required after reading an $\overline{a}$.

◁

Instead of directly constructing a DFA from a given DTD, we extend and adapt two algorithms for learning regular languages to this setting. The main objective is to provide a learning algorithm that yields for a recognizable DTD d a DFA $\mathcal{A}$ such that

$$L(d) = L^{XML} \cap L(\mathcal{A}) \ .$$

To develop such a learner, we consider a more general setting in which a regular language with "don't cares" is learned. In this context, a MAT does no longer answers membership queries either positively or negatively but also with "don't care" denoted by "?". An equivalence query also has to respect "don't cares" as it has to check whether the provided conjecture is a DFA that can have an arbitrary behavior on "don't cares" but needs to work correctly on all other words. A learning algorithm for regular languages with "don't cares" can then directly be applied to the current setting.

For the practical part of this thesis a MAT for XML word representations is needed, which can answer membership and equivalence queries for a DTD. We describe a straightforward construction method in Section 4.1. In Section 4.2 we present heuristic methods for learning regular languages with "don't cares" and their application to the setting of validating XML word representations. Finally, in Section 4.3 we present our practical work on learning algorithms: A proof-of-concept implementation of our learners and our MAT for XML word representations. We finish this section with a detailed analysis of the learner's behavior on some sample DTDs.

4.1 A minimally adequate teacher for XML word representations

It is unknown whether the decision problem for XML word representations

"Given a DTD d. Is d recognizable?"

is decidable. Nevertheless, Segoufin and Vianu [SV02] were able to find a precise characterization of recognizability when the given DTDs are "fully recursive", i.e. all tags that lead to recursive tags are mutually recursive. They were, moreover, able to provide an algorithm for constructing from a fully recursive and recognizable DTD a *standard DFA* that exactly accepts all valid XML word representations. For DTDs that are not fully recursive Segoufin and Vianu were able to identify several necessary conditions for recognizability. However, since it is clearly impossible to learn a DFA if the given DTD is not recognizable, we restrict ourselves in this chapter on recognizable DTDs. For the rest of this section let $d = (\Sigma_{tag}, r, \rho)$ be a recognizable DTD and $\Sigma = \Sigma_{tag} \cup \bar{\Sigma}_{tag}$.

For our proof-of-concept implementation (cf. Section 4.3), a MAT for XML word representations is needed. However, it is not obvious how such a MAT can be constructed. This section is, therefore, intended to present our theoretical implementation of a MAT for XML word representations.

In this context, the definitions of membership and equivalence queries need to be changed to fit in the new setting.

- A membership query is the check of a word $w \in \Sigma^*$ to be valid w.r.t. d. Thereby, the validity of a word over Σ is shifted from the validity of Σ_{tag}-valued trees. To be precise, we call w *valid* if $w \in L(d)$. If $w \in L^{XML}$ but $w \notin L(d)$, then w is called *not valid*. Since we do not care about words that are no XML word representations, we call w a *don't care* if $w \notin L^{XML}$.

  Accordingly, a MAT has to reply "yes" to a membership query if the input is valid w.r.t. d and "no" if the input is not valid. Furthermore, the MAT has to indicate that a given word is no valid XML word representation. In this case, the MAT replies with "don't care" or "?".

- An equivalence query is the check whether a given DFA $\mathcal{A}$ accepts all XML word representations that are valid w.r.t. d and rejects all other XML word representations. Furthermore, the MAT has to ignore the behavior of $\mathcal{A}$ on words that are no valid XML word representations. Formally, the MAT has to check whether

  $$L(d) = L^{XML} \cap L(\mathcal{A})$$

  holds. If this condition is satisfied, i.e. $\mathcal{A}$ is equivalent to d, then the MAT replies with "yes". Otherwise, the MAT has to provide a counter-example $w \in L^{XML}$ such that $w \in L(\mathcal{A}) \Leftrightarrow w \notin L(d)$ as a witness that $\mathcal{A}$ and d are not equivalent.

4.1.1 Validating Σ-valued trees

Before we describe the function of a MAT for XML word representations, we have to do one step back and show formally how a Σ_{tag}-valued tree can be validated against a given DTD. This sharpens the informal intuition given in the introduction and helps to gain a deeper understanding of the validation of XML word representations.

While working with DTDs, it turns out that the use of horizontal language DFAs is more appropriate than the use of the regular expressions of a right hand side of a rule. Thus, let $\mathcal{A}_a = (Q_a, \Sigma_{tag}, q_0^a, \delta_a, F_a)$ be the horizontal language DFA for all $a \in \Sigma_{tag}$.

We observe that the validity of a Σ_{tag}-valued tree $t = (dom_t, val_t)$ w.r.t. d can be checked locally using the following condition $(\star)$:

1. For an inner node $w \in dom_t$ of t labeled with $val_t(w) = a$, we have to check whether the rule $\rho(a)$ was applied correctly. In other words, we have to check whether the labels $val_t(w1) \dots val_t(wn)$ of the ordered sequence $w1 < \ldots < wn$ of the sons of w form a word that is in $L(\mathcal{A}_a)$.
2. If w is a leaf node and, therefore, has no children, then the empty sequence has to be accepted by $\mathcal{A}_a$, i.e. $\varepsilon \in L(\mathcal{A}_a)$.

Note that, if the local condition $(\star)$ is satisfied by all nodes of t, then the tree is valid w.r.t. d.

Example 4.3. For a better understanding of the condition $(\star)$, consider the DTD d in Figure 4.2(a) and the Σ-valued tree t in Figure 4.2(b). To validate the tree t w.r.t. d, we have to check $(\star)$ for each node. Let us consider two nodes of t in detail:

- Let $w = 1$, whose sons $11, 12$ and 13 are marked by the oval in Figure 4.2(b). Since w is labeled with a, we have to check whether the word $val_t(11)val_t(12)val_t(13) = bbb$ is in $L(\mathcal{A}_a)$.
- Let $w' = 12$. According to Figure 4.2(b), the node is labeled with b. Since w' is a leaf node, $children_t(w') = \emptyset$ and we have to check whether $\varepsilon \in L(\mathcal{A}_b)$.

We observe that in the setting of Figure 4.2 all nodes satisfy $(\star)$. Thus, t is valid w.r.t d.

$\triangleleft$

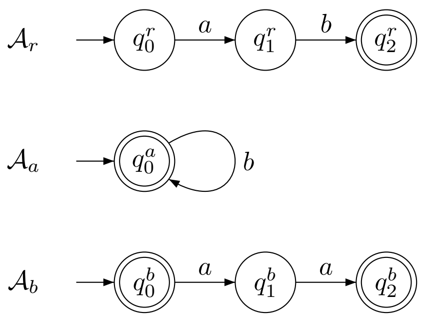
However, it is not obvious how to check the condition $(\star)$. Our idea is to label the tree t with states of the DFAs $\mathcal{A}_a$ for the horizontal languages of d. We do this by defining a mapping $\rho_{d,t} : \ dom_t \setminus \{\varepsilon\} \to \bigcup_{a \in \Sigma_{tag}} Q_a$ and label the tree as follows: For every inner node $w \in dom_t$ labeled with $val_t(w) = a$, we assign states from $\mathcal{A}_a$ to the ordered sequence of its sons $w1 < \ldots < wn$. Moreover, we require that this sequence of states, together with the labels of $w1, \ldots, wn$, produces a run of $\mathcal{A}_a$ starting in the initial state q_0^a, i.e.

$$q_0^a \xrightarrow{val_t(w1)}_{\mathcal{A}_a} \rho_{d,t}(w1) \xrightarrow{val_t(w2)}_{\mathcal{A}_a} \cdots \xrightarrow{val_t(wn)}_{\mathcal{A}_a} \rho_{d,t}(wn) \ .$$

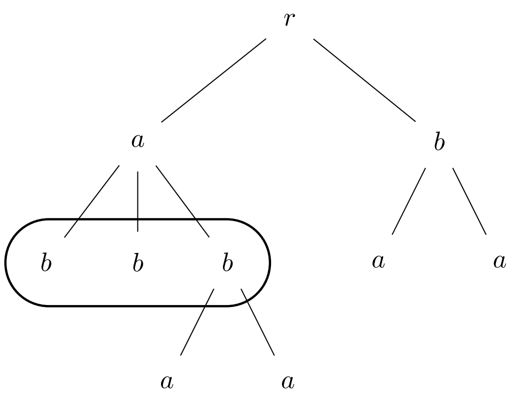
The state assigned to the node wi is determined by the state of its left brother $w(i-1)$, the label of wi and δ_a, namely $\rho_{d,t}(wi) = \delta_a(\rho_{d,t}(w(i-1)), val_t(wi))$, where $i = 2, \ldots, n$. Since the node $w1$ has no left brother, its assigned state is determined by $\delta_a(q_0^a, val_t(w1))$.

Note that the first state q_0^a has to be added as the initial state of the run and that all sons of w have to be labeled with states of the same DFA $\mathcal{A}_a$. Furthermore, note that the root node ε is labeled in no case.

$$
d: \quad \begin{array}{l} r \to ab \\ a \to b^* \\ b \to \varepsilon \mid aa \end{array}
$$



(a) The DTD d and the associated DFAs $\mathcal{A}_r$, $\mathcal{A}_a$ and $\mathcal{A}_b$



(b) The Σ-valued tree t

Figure 4.2: The DTD d and the Σ-labeled tree t of Example 4.3

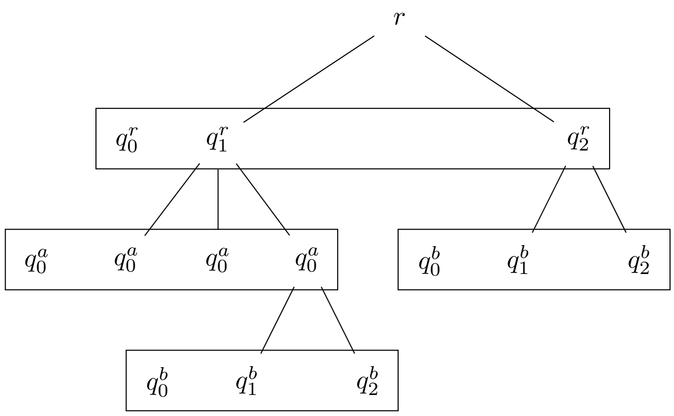


Figure 4.3: A run of the DTD d on the tree t of Example 4.3

We now check whether the rightmost son of each inner node is labeled with an accepting state. In this case, the run coded by $\rho_{d,t}$ is an accepting run and $val_t(w1)\ldots val_t(wn) \in L(\mathcal{A}_a)$.

Moreover, we have to check the second part of condition $(\star)$, i.e. whether $\varepsilon \in L(\mathcal{A}_{val_t(w)})$ for all leaf nodes $w \in dom_t$. This can be done by checking whether $q_0^a \in F_a$ if $val_t(w) = a$.

If these requirements are satisfied by all nodes of t, then the tree t is valid w.r.t. d.

Example 4.4. We use d and t from Example 4.3 to make our idea clear. A labeling $\rho_{d,t}$ of t is shown in Figure 4.3. The boxes indicate which nodes form a run of a horizontal language DFA.

Since in this example only accepting states are assigned to rightmost sons of inner nodes and $\varepsilon \in L(\mathcal{A}_a)$ for all leaf nodes labeled with a, the tree t is valid w.r.t. d.

$\triangleleft$

We formalize our idea in the following definition.

Definition 4.5. [Run of a DTD on a Σ_{tag}-valued tree] The *run* of a DTD $d = (\Sigma_{tag}, r, \rho)$ on a Σ_{tag}-labeled tree $t = (dom_t, val_t)$ is a mapping

$$\rho_{d,t} :\ dom_t \setminus \{\varepsilon\} \to \bigcup_{a \in \Sigma_{tag}} Q_a$$

that satisfies the following condition.

For an inner node $w \in dom_t$ labeled with $vat_t(w) = a$ let $w1 < \ldots < wn$ for $n = rank_t(w)$ be the ordered sequence of all children of w. Then,

$$q_0^a \xrightarrow{val_t(w1)}_{\mathcal{A}_a} \rho_{d,t}(w1) \xrightarrow{val_t(w2)}_{\mathcal{A}_a} \ldots \xrightarrow{val_t(wn)}_{\mathcal{A}_a} \rho_{d,t}(wn)$$

is a run of $\mathcal{A}_a$ on the word $val_t(w1)\ldots val_t(wn)$.

A run is said to be *accepting* if and only if

1. $val_t(\varepsilon) = r$,
2. for all $w \in dom_t$ with $children_t(w) \neq \emptyset$ the state of the rightmost son wn with $n = rank(w)$ is accepting, i.e. $\rho_{d,t}(wn) \in F_{val_t(w)}$, and
3. for all $w \in dom_t$ with $children_t(w) = \emptyset$ the condition $q_0^{val_t(w)} \in F_{val_t(w)}$ holds.

◁

Note that there is at most one run of d on a Σ_{tag}-valued tree t. If t is consisting of a single root node and $\rho_{d,t}$ is the run of d on t, then $\rho_{d,t}$ is the empty mapping. By definition $\rho_{d,t}$ is accepting if and only if the root node is labeled with r and $\varepsilon \in L(\mathcal{A}_r)$.

With this notation we can now formally define the validity of a Σ_{tag}-valued tree w.r.t. a DTD.

Definition 4.6. A Σ_{tag}-labeled tree t is *valid* w.r.t. a DTD d if and only if there is an accepting run $\rho_{d,t}$ of d on t.

◁

4.1.2 Answering queries

Recall that DTDs basically represent context free grammars. However, automata are more appropriate than grammars when dealing with words. We, therefore, use an automata based approach to construct a minimally adequate teacher for XML word representations. For this purpose we use the well known fact that each context free grammar is equivalent to a pushdown automaton and vice versa.[1]

The validation of XML word representations can already be done by an extended visibly pushdown automaton as introduced in Section 2.3 (cf. Definition 2.16). Such an eVPA needs its stack only to store information about the path to the current position within the represented tree. Since eVPAs are sufficient and enjoy nice properties, it is reasonable to use them instead of pushdown automata.

The MAT for a DTD $d = (\Sigma_{tag}, r, \rho)$ is based on an eVPA, denoted by $\mathcal{A}_d$, that processes Σ_{tag}-valued trees coded as XML word representations and accepts exactly all valid XML word representations, i.e. $L(\mathcal{A}_d) = L(d)$. Furthermore, it can distinguish between XML word representations and inputs that are none. In this setting, the set of opening tags represents the calls while the set of closing tags represents the returns. This means $\Sigma_c = \Sigma_{tag}$, $\Sigma_r = \bar{\Sigma}_{tag}$ and $\Sigma_{int} = \emptyset$. The function of the MAT is sketched in Figure 4.4.

For reasons of simplicity we first restrict ourselves to show how $\mathcal{A}_d$ performs the validation of well-formed XML word representations. The treatment of words not in L^{XML} is done later on. Thus, let $w = [t] \in L^{XML}$ be the XML word representation of a Σ-valued tree t.

A useful observation about XML word representations is that they are depth-first traversals of the corresponding Σ_{tag}-valued trees. Each node of the tree is

[1] See [HMU01] for more information regarding the equivalence between context free grammars and pushdown automata

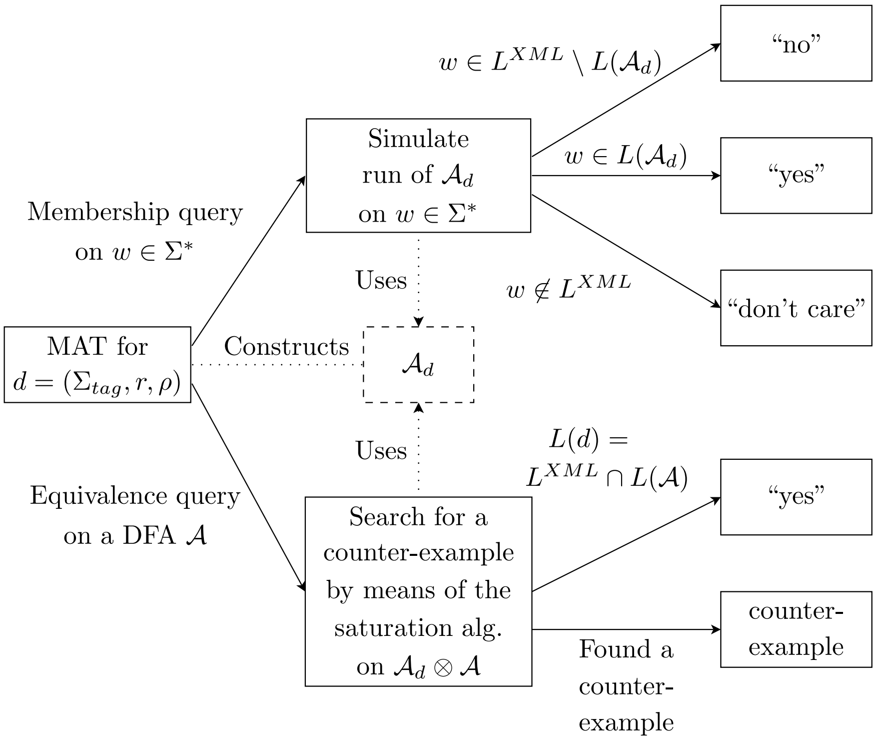


Figure 4.4: View of the function of a MAT for XML word representations

represented by two tags, namely one opening tag and the unique matching closing tag. In our setting, this is the corresponding call and the unique matching return.

An eVPA can exploit this property. Since an eVPA has to push a symbol on the stack every time it reads an opening tag and has to pop a symbol while reading a closing tag, the stack is a representation of the path from the currently processed node in the XML word representation to the root node.

Thus, our idea is to use the stack of the eVPA $\mathcal{A}_d$ to successively compute a run $\rho_{d,t}$ of a DTD d on the Σ-valued tree t. Thereby, we use the stack to represent the run from the node currently processed to the root node. In particular, the stack consists of the states of the horizontal language DFAs on this path. Thus, every time $\mathcal{A}_d$ reads a new symbol it has to update the stack to compute a new part of the run $\rho_{d,t}$.

With this intuition we design the construction of $\mathcal{A}_d$ such that it results in an eVPA that preserves the following constraints ($\Diamond$):

1. After reading the opening tag a of a node $w = a_1 \ldots a_n$ with $val_t(w) = a$, the stack content is

$$q_0^a \rho_{d,t}(a_1 \ldots a_n)\rho_{d,t}(a_1 \ldots a_{n-1}) \ldots \rho_{d,t}(a_1)\bot \ .$$

2. After reading the closing tag $\bar{a}$ of the node w, the stack content is

$$\rho_{d,t}(a_1 \ldots a_n)\rho_{d,t}(a_1 \ldots a_{n-1}) \ldots \rho_{d,t}(a_1)\bot \ .$$

We archive this as follows. Whenever $\mathcal{A}_d$ reads an opening tag $a \in \Sigma_{tag}$, it replaces the topmost stack symbol $q^b \in Q_b$ with $\delta_b(q^b, a)$ to simulate the run of the horizontal automaton $\mathcal{A}_b$. Furthermore, it pushes the state q_0^a on the stack. If $\mathcal{A}_d$ reads a closing tag $\bar{a} \in \bar{\Sigma}_{tag}$, it simply pops the top stack symbol q^a from the stack. Since this symbol represents the state of the rightmost son of the current node, $\mathcal{A}_d$ checks whether $q^a \in F_a$ or not. In the first case, the automaton proceeds with the next symbol. In the latter case, the word gained from the labels of the sons of the current node is not accepted by $\mathcal{A}_a$ and, thus, the computed run of d is not accepting. As we see later, we also need to decide whether the input is an XML word representation that is not valid w.r.t. d. Since there is also a run of d on such trees (which is of course not accepting), we store the information that t is not valid but still an XML word representation in the states of the automaton and proceed.

Let us now consider words $w \in \Sigma^* \setminus L^{XML}$ that are no valid XML word representations. An input can be no XML word representation due to some reasons:

- w is no well-matched word. Then, there is no run of any eVPA on w or the run does not end in an accepting configuration. In both cases, we know that w is no XML word representation.

- w is a well-matched word but of the form $w = w_1 \dots w_n$ with $w_i \in L_{wm}$ for $i = 1, \dots, n$. We have to ensure that such words are not accepted by $\mathcal{A}_d$. We overcome this problem as we only allow $\mathcal{A}_d$ to read the stack bottom symbol $\perp$ in the first step and then never again.

- w is well matched and no concatenation of well-matched words (as above) but no valid XML word representation. Then, there is at least one opening tag a whose matching closing tag is not $\bar{a}$. The automaton $\mathcal{A}_d$ can check for such situations whenever it reads a return symbol $\bar{a}$. Since the topmost stack symbol q^x indicates the label of the current node of t, it suffices to check if the input is the associated closing tag to x, i.e. to check whether $x = a$. We construct $\mathcal{A}_d$ such that there is no transition to apply in the case that $x \neq a$.

Before we give the formal construction of $\mathcal{A}_d$, we briefly describe the implementation of the above given intuition.

Since most of the work of $\mathcal{A}_d$ is done using the stack, the three states q_0, q and q_t suffice. The state q_0 is the unique initial state. It is the only state in which $\mathcal{A}_d$ can perform a transition on the empty stack. Moreover, it can reach q_0 never again. This ensures that $\mathcal{A}_d$ does not accept words that are sequences of well-matched words.

From q_0 the automaton can read r, if r is the root symbol, and change its state to q. The state q is used to simulate the run $\rho_{d,t}$. If the input w codes a tree t valid w.r.t. d, i.e. the input is in $L(d)$, the automaton computes an accepting run of d on t and ends in state q with the empty stack. Since we set q as the only final state, the input is then accepted. If $\mathcal{A}_d$ discovers the input to be no XML word representation, it simply stops reading the input. In this case, our construction ensures that there is no applicable transition. If w is an XML word representation but not valid w.r.t. d, then the automaton changes its control state to q_t and tries to compute a non-accepting run of d on t. The automaton recognizes this situation once it reads a closing $\bar{a}$ but the top stack symbol is a state $q^a \notin F_a$ or if the first symbol read is not the root symbol r.

The state q_t indicates that w is a valid XML word representation so far but not valid w.r.t. d. From this state $\mathcal{A}_d$ cannot reach any other state. However, the same actions can be performed as in state q. If the run of $\mathcal{A}_d$ on w ends in q_t with the empty stack, we know that $w = [t]$ for a tree t that is not valid w.r.t. d.

Let us now present the formal construction of the automaton $\mathcal{A}_d$.

Construction 4.7. *For a DTD $d = (\Sigma_{tag}, r, \rho)$ and its horizontal language DFAs $\mathcal{A}_a = (Q_a, \Sigma_{tag}, q_0^a, \delta_a, F_a)$ for all $a \in \Sigma_{tag}$, we define the eVPA $\mathcal{A}_d = (Q_d, W_{in}^d, \Sigma, \Gamma_d, \Delta_d, F_d)$ as*

- $Q_d = \{q_0, q, q_t\}$,
- $Q_{in}^d = \{q_0\}$,
- $\Gamma_d = \bigcup_{a \in \Sigma_{tag}} Q_a$,
- $F_d = \{q\}$ *and*
- *the transition relation defined by*

$$\begin{array}{l}
(q_0, r, \bot, q_0^r\bot, q) \in \Delta_d \ , \\
(q_0, a, \bot, q_0^a\bot, q_t) \in \Delta_d \textit{ for } a \in \Sigma_{tag},\ a \neq r \ , \\
\\
(q, a, q^b, q_0^a p^b, q) \in \Delta_d \textit{ for } a, b \in \Sigma_{tag},\ q^b \in Q_b \textit{ and } \delta_b(q^b, a) = p^b \ , \\
(q, \bar{a}, q^a, \varepsilon, q) \in \Delta_d \textit{ for } a \in \Sigma_{tag} \textit{ and } q^a \in F_a \ , \\
(q, \bar{a}, q^a, \varepsilon, q_t) \in \Delta_d \textit{ for } a \in \Sigma_{tag} \textit{ and } q^a \notin F_a \ , \\
\\
(q_t, a, q^b, q_0^a p^b, q_t) \in \Delta_d \textit{ for } a, b \in \Sigma_{tag},\ q^b \in Q_b \textit{ and } \delta_b(q^b, a) = p^b \ , \\
(q_t, \bar{a}, q^a, \varepsilon, q_t) \in \Delta_d \textit{ for } a \in \Sigma_{tag} \ .
\end{array}$$

Note that $\mathcal{A}_d$ is deterministic in the sense it has a unique initial state and that every configuration has at most one a-successor.

Theorem 4.8. *Construction 4.7 yields an eVPA $\mathcal{A}_d$ with $L(\mathcal{A}_d) = L(d)$.*

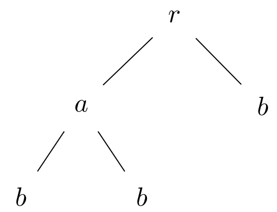
Proof. It is not hard to verify that $\mathcal{A}_d$ preserves the following properties for an input $w \in \Sigma^*$.

- If $w \in L(d)$, then
$$(q_0, \bot) \xrightarrow{w}_{\mathcal{A}_d} (q, \bot) \ .$$
- If $w \notin L(d)$ but $w \in L^{XML}$, then
$$(q_0, \bot) \xrightarrow{w}_{\mathcal{A}_d} (q_t, \bot) \ .$$
- Finally, if $w \notin L^{XML}$, then there is no run of $\mathcal{A}_d$ on w or the run does not end up with the empty stack.

A formal proof can be done by induction using the conditions ($\Diamond$) on page 52. However, we skip the details on this straightforward induction.

□

Example 4.9. Consider the DTD d from Example 4.3 and the following Σ_{tag}-valued tree t:



The XML word representation of t is $[t] = rab\bar{b}b\bar{b}\bar{a}b\bar{b}\bar{r}$. A run of $\mathcal{A}_d$ on $[t]$ is given by

$$\begin{aligned} & (q_0, \bot) \xrightarrow{r} (q, q_0^r\bot) \xrightarrow{a} (q, q_0^a q_1^r\bot) \xrightarrow{b} (q, q_0^b q_0^a q_1^r\bot) \xrightarrow{\bar{b}} (q, q_0^a q_1^r\bot) \\ \xrightarrow{b} \; & (q, q_0^b q_0^a q_1^r\bot) \xrightarrow{\bar{b}} (q, q_0^a q_1^r\bot) \xrightarrow{\bar{a}} (q, q_1^r\bot) \xrightarrow{b} (q, q_0^b q_2^r\bot) \\ \xrightarrow{\bar{b}} \; & (q, q_2^r\bot) \xrightarrow{\bar{r}} (q, \bot) \ . \end{aligned}$$

Since $(q_0, \bot) \xrightarrow{[t]} (q, \bot)$ and $q \in F$, the XML word representation of t is accepted by $\mathcal{A}_d$ and, therefore, valid w.r.t. d, i.e. $[t] \in L(d)$.

◁

Answering membership queries

By using the automaton $\mathcal{A}_d$, it is now easy for a MAT to answer membership queries for a given input $w \in \Sigma^*$. The MAT simply computes the run of $\mathcal{A}_d$ on w. If the run ends in the configuration $(q, \bot)$, then $w \in L(d)$ and the MAT returns "yes". If the run ends in a configuration $(q_t, \bot)$, then $w \in L^{XML}$ but $w \notin L(d)$. In this case, the MAT returns "no". In any other case, the MAT returns "don't care".

As a final remark, let us have a closer look at the complexity of the construction.

Remark 4.10. The eVPA $\mathcal{A}_d$ has only 3 states. Furthermore, it has $\sum_{a \in \Sigma_{tag}} |Q_a|$ many stack symbols and $\mathcal{O}(\sum_{a \in \Sigma_{tag}} |Q_a| \cdot |\Sigma_{tag}|)$ many transition. Since we assume the DTD to be given, the construction of $\mathcal{A}_d$ only needs to be done once.

The validation of a word $w \in \Sigma^*$ and, thus, the answer of a membership query, can be done in time linear in the length of w, i.e. in $\mathcal{O}(|w|)$.

◁

Answering equivalence queries

Let us recall the task of answering equivalence queries. The MAT is provided with a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ and has to check

1. if $\mathcal{A}$ accepts all XML word representations that are valid w.r.t. d and
2. if $\mathcal{A}$ rejects all XML word representations that are not valid w.r.t. d.

Equivalently, we can check whether there is an XML word representation $w \in \Sigma^{XML}$ with $w \in L(d) \Leftrightarrow w \notin L(\mathcal{A})$. Recall that the MAT has to ignore the behavior of $\mathcal{A}$ on words which are no XML word representations.

We use the eVPA $\mathcal{A}_d = (Q_d, \Sigma, \{q_0^d\}, \Gamma_d, \Delta_d, F_d)$ of Section 4.1.2 to check whether a word w is an XML word representation and if it is valid w.r.t. d. If w is no XML word representation, we know that there is no run of $\mathcal{A}_d$ on w or the run ends in a configuration where the stack is not empty. As mentioned above, we do not care about such words. If w is an XML word representation, then the run of $\mathcal{A}_d$ on w ends in the configuration $(q, \perp)$ or $(q_t, \perp)$ depending on whether w is valid w.r.t. d or not. In the first case, the run ends in state q while in the latter case the run ends in state q_t.

With this in mind, the conditions 1. and 2. above are violated and, thus, $\mathcal{A}$ is not equivalent to d if there is a word w such that

1. $(q_0^d, \perp) \xrightarrow{w}_{\mathcal{A}_d} (q, \perp)$ and $q_0 \xrightarrow{w}_{\mathcal{A}} q'$ with $q' \notin F$ or

2. $(q_0^d, \perp) \xrightarrow{w}_{\mathcal{A}_d} (q_t, \perp)$ and $q_0 \xrightarrow{w}_{\mathcal{A}} q'$ with $q' \in F$.

The idea how to check these conditions is to construct the product $\mathcal{A}_d \otimes \mathcal{A}$ (cf. Definition 2.24), which simulates both $\mathcal{A}_d$ and $\mathcal{A}$ simultaneously. Then, we check whether a configuration $((q, p), \perp)$ with $p \notin F$ or $((q_t, p'), \perp)$ with $p' \in F$ is reachable from the unique initial configuration $((q_0^d, q_0), \perp)$. Thereto, we define a regular set C of configurations as

$$C = \{((q, p), \perp) \mid p \notin F\} \cup \{((q_t, p), \perp) \mid p \in F\}$$

and compute an NFA $\bar{\mathcal{B}}$ for $pre^*(C)$ by using the extended saturation algorithm presented in Section 2.3. It remains to check whether $\bar{\mathcal{B}}$ accepts $\perp$ from state (q_0^d, q_0) and to compute a counter-example w if so.

To answer equivalence queries, a MAT performs the steps described and returns

- "yes" if $\bar{\mathcal{B}}$ rejects $\perp$ from state (q_0^d, q_0) since $\mathcal{A}$ and d are equivalent or
- the counter-example w gained from an accepting run of $\bar{\mathcal{B}}$ on $\perp$ starting in state (q_0^d, q_0) as a witness that $\mathcal{A}$ and d are not equivalent.

However, this procedure does not guarantee to find a minimal counter-example. To compute a minimal counter-example, we can use the two-tier approach also presented in Section 2.3. Thereto, we first have to check whether a counter-example exists as described above. Then, we perform a backwards breadth-first search in the set of all configurations that can reach a configuration in C.

As a last remark, we consider the complexity of an equivalence query.

Remark 4.11. The construction of the product $\mathcal{A}_d \otimes \mathcal{A}$ results in an eVPA, which has

- $|Q_{\mathcal{A}_d \otimes \mathcal{A}}| = 3 \cdot |Q|$ states,
- $|\Gamma_{\mathcal{A}_d \otimes \mathcal{A}}| = \sum_{a \in \Sigma_{tag}} |Q_a|$ stack symbols and
- $|\Delta_{\mathcal{A}_d \otimes \mathcal{A}}| = \left(\sum_{a \in \Sigma_{tag}} |Q_a| \cdot |\Sigma_{tag}|\right) \cdot |Q|$ transitions.

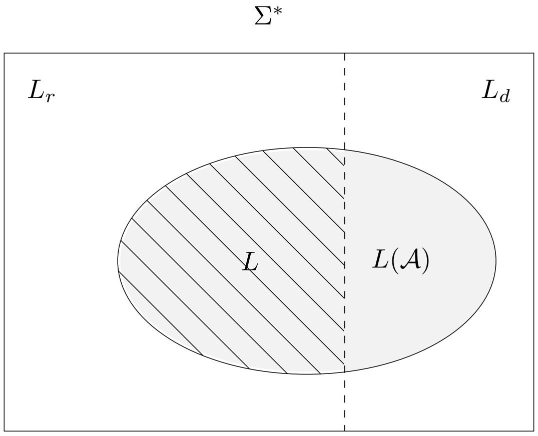


Figure 4.5: The setting of learning regular languages with "don't cares". The shaded area represents the language L while the gray oval represents the language $L(\mathcal{A})$

Thus, the construction of $Q_{\mathcal{A}_d \otimes \mathcal{A}}$ can be done in $\mathcal{O}(|Q|)$ since the DTD d and, hence, the eVPA $\mathcal{A}_d$ is fixed.

The P-automaton $\mathcal{B}$ has $|Q_\mathcal{B}| = |Q_{\mathcal{A}_d \otimes \mathcal{A}}| + 1$ states. Since the saturation algorithm runs in time $\mathcal{O}(|\Delta_{\mathcal{A}_d \otimes \mathcal{A}}| \cdot |\Gamma_{\mathcal{A}_d \otimes \mathcal{A}}| \cdot |Q_{\mathcal{A}_d \otimes \mathcal{A}}| \cdot |Q_\mathcal{B}|^3)$, this yields a runtime of $\mathcal{O}(|Q|^5)$. Thereafter, a counter-example w can be computed in time $\mathcal{O}(|w|)$ which yields an overall time complexity of $\mathcal{O}(|Q|^5 + |w|)$. However, the runtime of the breadth-first search for a shortest counter-example can be exponential in the size of w.

$\triangleleft$

4.2 Heuristic methods for learning regular languages with "don't cares"

In the setting of regular languages with "don't cares" the set Σ^* of all inputs is disjointly divided into the set L_d of *"don't cares"* and the set $L_r = \Sigma^* \setminus L_d$ of *relevant inputs*. The pair (L, L_r) with $L \subseteq L_r$ is called a *regular language with "don't cares"* if there exists a DFA $\mathcal{A}$ that agrees with L on set of relevant inputs but may has arbitrary behavior on "don't cares". Formally, we require that

$$L = L_r \cap L(\mathcal{A})$$

holds for the DFA $\mathcal{A}$. In this case, we say that $\mathcal{A}$ is *equivalent* to (L, L_r). If L_r is clear from the context, we refer to (L, L_r) simply as L. This situation is depicted in Figure 4.5. Note that the language L itself does not need to be regular.

In this context, the general objective of learning algorithms is to identify a DFA that is equivalent with an a priori fixed regular language with "don't cares". Following Grinchtein and Leucker [GL], several setups seem possible besides the one where we intentionally do not care about certain inputs.[2] For

[2] We can easily use a learner for regular languages with "don't cares" to learn a DFA $\mathcal{A}$ equivalent to a DTD d by setting $L = L(d)$ and $L_r = L^{XML}$

reasons of efficiency the MAT might not be able to compute the correct answer for every query or the underlying system is incompletely specified so that the MAT is obviously forced to return "?" to some queries. Furthermore, a MAT may be based on a class $\mathcal{C} = \{L_1, \ldots, L_n\}$ of languages over a fixed alphabet and returns either a positive or a negative answer to a membership query whenever the input is in all or in none of these languages respectively. In any other case, it returns "don't care".

Let us briefly describe the function of a MAT in the setting of learning regular languages with "don't cares".

- On a membership query the learner presents a word $w \in \Sigma^*$. The MAT replies "yes" if $w \in L$ and "no" if $w \in L_r \setminus L$. Moreover, the MAT indicates that the input is a "don't care" by returning "?" if $w \in L_d$.

- On an equivalence query the learner provides a conjecture $\mathcal{A}$ and asks whether
$$L = L_r \cap L(\mathcal{A})$$
holds. If this is the case, the MAT returns "yes". Otherwise, it returns a counter-example $w \in L(\mathcal{A}) \cap L_r \Leftrightarrow w \notin L$.

However, the difference between the task of learning regular languages and the one of learning of regular languages with "don't cares" is the fact that the target language is no longer unique. In fact, a DFA equivalent to (L, L_r) is free to accept or reject "don't cares". Thus, a learning algorithm has to be able to choose a reasonable conjecture's behavior on these words. The learner can also change this behavior during the learning process. The hardness of the problem lies therein.

Grinchtein and Leucker [GL] address a similar task, which they call "learning from inexperienced teachers". Their learning algorithm may also be applied to this setting but their results are not yet published.

In the following we describe our heuristic method to learn regular languages with "don't cares". By heuristic method, we mean a reasonable procedure that results in a correct solution but we do not know whether it always terminates. A proof of termination is not a part of this thesis. Instead we do practical work and implement the heuristics described in the following.

The first two heuristics , called $L^*_?$ and $L^*_{?\ V2}$, are based on Angluin's learner for regular languages. They are presented in Section 4.2.1. The third, presented in Section 4.2.2, relies on the RPNI algorithm. It is called RPNI$_?$. Finally, in Section 4.2.3 and 4.2.4 we consider some general improvements and one for the special case of learning DFAs for validating XML word representations.

4.2.1 Angluin's learner for regular languages with "don't cares"

Next, we describe our $L^*_?$ learner. It uses an observation table to store the information about the target language and uses it to construct a conjecture. However, in contrast to Section 3.1, a MAT does no longer respond either with "yes" or "no" to membership queries. It also answers with "?" if the provided word is a "don't care". To be able to deal with this special kind of answers, we extend the observation table accordingly. This is done in the following definition.

Definition 4.12. [Incomplete observation table] An *incomplete observation table* is a tuple $O = (R, S, T)$ consisting of

- a nonempty, finite, prefix-closed set $R \subseteq \Sigma^*$ of *representatives*,
- a nonempty, finite, suffix-closed set $S \subseteq \Sigma^*$ of *samples*
- and a mapping $T:\ (R \cup R \cdot \Sigma) \cdot S \to \{0, 1, ?\}$.

$\triangleleft$

We require (and ensure this during the run of the learner) that an incomplete observation table agrees with the MAT in the sense that $T(w) =? \Leftrightarrow w \in L_d$, $T(w) = 1 \Leftrightarrow w \in L \cap L_r$ and $T(w) = 0 \Leftrightarrow w \in L_r \setminus L$ holds for all w in the domain of T.

Since our learner has to choose the conjecture's behavior on "don't cares", it can fit this behavior to its needs. Moreover, it can even change the conjecture's behavior during the learning process. The disadvantage of this freedom of choice is that it is no longer possible to check whether an incomplete observation table is closed or consistent since each ?-entry can be set to either 0 or 1. The same problem occurs while constructing a conjecture. Thus, we have to decide which value we assign to a ?-entry before every such operation.

Since we can assign either 0 or 1 to a ?-entry, we can think of these entries as Boolean variables x_w. Then, the closed and consistent conditions can be expressed as Boolean formulae φ_{cl} and φ_{co} over these variables. If we find a model μ that satisfies these formulae, then this assignment produces a closed and consistent observation table. Moreover, we can use μ to construct a conjecture.

To do so, we define $O_\mu = (R, S, T_\mu)$ by replacing the mapping T in the incomplete observation table O with T_μ. Thereby, T_μ replaces all ?-entries in T with the values of the model μ. Formally, T_μ is defined as

$$T_\mu:\ (R \cup R \cdot \Sigma) \cdot S \to \{0, 1\} \ \text{ with } \ T_\mu(w) = \begin{cases} T(w) & \text{, if } T(w) \neq ? \\ \mu(x_w) & \text{, if } T(w) = ? \end{cases}$$

for all $w \in (R \cup R \cdot \Sigma) \cdot S$. Note that $O_\mu = (R, S, T_\mu)$ is now an observation table.

To formulate φ_{cl} and φ_{co}, it is useful to represent every entry $w \in (R \cup R \cdot \Sigma) \cdot S$ of the incomplete observation table by a Boolean variable x_w even if it is not a ?-entry. There are two simple possibilities how to ensure that a SAT solver assigns the value stored in the table to a variable x_w if $T(w) \neq ?$:

1. We extend φ_{cl} and φ_{co} to $\varphi'_{cl} = \varphi_{cl} \wedge \varphi_T$ and $\varphi'_{co} = \varphi_{co} \wedge \varphi_T$ by adding a formula φ_T. Thereby, the formula φ_T forces $x_w = T(w)$ and is defined as

$$\varphi_T = \bigwedge_{w \in (R \cup R \cdot \Sigma) \cdot S,\ T(w) \neq ?} I(w)$$

where

$$I(w) = \begin{cases} \neg x_w & \text{, if } T(w) = 0 \\ x_w & \text{, if } T(w) = 1 \end{cases} .$$

2. We replace syntactically every occurrence of the variable x_w with the Boolean constant $T(w)$ before the formulae are solved.

However, we prefer the second procedure since it reduces the number of variables and does not change the size of the formulae.

Before we describe how the closed and consistent formulae are constructed, let us briefly sketch the main idea of the $L^*_?$ learner. The $L^*_?$ heuristic basically works in the same way as Angluin's original learner. The overall goal is to compute a conjecture that is compatible with the data stored in the incomplete observation table. This is done by computing values for the ?-entries before each operation by finding models for the formula $\varphi_{cl} \wedge \varphi_{co}$. Of course, we also have to deal with the situation in which there is no such model.

The models are computed by means of a SAT solver. Since the behavior and results of a SAT solver are not completely predictable, we understand it as a black box. Unfortunately, solving SAT instances is known to be NP-complete [Coo71, Pap95] and even if today's SAT solvers are highly sophisticated and allow to solve huge instances, there is little hope that this can be done efficiently (unless $P = NP$). However, the next theorem shows that finding a closed and consistent observation table by means of solving SAT instances is reasonable since the problem itself is NP-complete.

Theorem 4.13 (Gold [Gol78])**.** *Computing an assignment of Boolean values to ?-entries such that an incomplete observation table is closed and consistent is NP-complete.*

Gold [Gol78] studied a rather similar problem, which he calls *hole-filling problem*. To use our terminology, in the hole-filling problem an incomplete observation table with ?-entries, which he calls *holes*, is given. The task is to find an assignment of Boolean values to the holes that yields a closed table. Gold was able to prove that the hole-filling problem is NP-complete by reducing the satisfiability problem for CNF formulae, which is well known to be NP-complete, to the hole-filling problem. The idea of this reduction is to construct an incomplete observation table that has a row for each clause and a column for each literal (for technical reasons there are some more rows and columns) such that, if a literal occurs in a certain clause, then the corresponding table entry is a hole. Moreover, Gold's construction ensures that there is a solution for the hole-filling problem if and only if there is a model for the CNF formula. With this result Gold was also able to prove that a similar problem, namely identifying a minimal NFA compatible with a given set of words, is NP-complete.

Using Gold's results, it is not hard to prove Theorem 4.13.

Proof of Theorem 4.13. Gold's proof of the NP-completeness of the hole filling problem directly yields this result. Gold's reduction ensures that each representative has a unique row. Thus, any assignment that makes the table closed also makes it consistent. Therefore, finding an assignment that makes the table closed and consistent has to be NP-complete, too.

□

We, therefore, try to keep the size of the formulae as small as possible. Moreover, their size is of special interest in connection with the input format of SAT solvers. Most SAT solvers only accept SAT instances that are in conjunctive normal form. This requires that an arbitrary SAT instance first has to be transformed into CNF before it can be solved. However, the naive translation into CNF using De Morgan's and the distributive law generally results in a formula, which has exponentially many clauses. Thus, a more efficient translation is essential.

In the following let $O = (R, S, T)$ be an incomplete observation table and $x_w \in \{0, 1\}$ be Boolean variables for all entries $w \in (R \cup R \cdot \Sigma) \cdot S$ of the table. Moreover, let $x_{w_1}, \ldots, x_{w_n}$ be an enumeration of all these Boolean variables.

Closed formula

Let us briefly recall the meaning of the closed condition. An observation table is closed if and only if for every representative $u \in R \cdot \Sigma$ its equivalence class $[\![u]\!]_O$ also has a representative v in R. In other words there has to be a representative $v \in R$ that has the same row as the representative u. We can express this formally by the following first order formula:

$$\forall u \in R \cdot \Sigma \; \exists v \in R : \; \forall w \in S : \; T(u \cdot w) = T(v \cdot w) \; .$$

It is not difficult to translate this formula into a Boolean formula φ_{cl} over the variables x_w.

$$\varphi_{cl} = \bigwedge_{u \in R \cdot \Sigma} \; \bigvee_{v \in R} \; \bigwedge_{w \in S} \Big[(x_{u \cdot w} \wedge x_{v \cdot w}) \vee (\neg x_{u \cdot w} \wedge \neg x_{v \cdot w}) \Big]$$

The naive transformation into CNF using the distributive law produces a CNF formula, which is exponential in the size of the original formula. To avoid this blow-up, we use Boolean auxiliary variables $y_{u,v}$ for $u \in R \cdot \Sigma$ and $v \in R$ with the following meaning:

> "The auxiliary variable $y_{u,v}$ is set to 1 if and only if the row of u and the row of v are equal."

We can now rewrite the closed formula as

$$\begin{aligned} \psi_{cl} \quad = \quad & \bigwedge_{u \in R \cdot \Sigma} \bigvee_{v \in R} y_{u,v} \\ \wedge \quad & \bigwedge_{u \in R \cdot \Sigma, v \in R} \left(y_{u,v} \rightarrow \bigwedge_{w \in S} [(x_{u \cdot w} \wedge x_{v \cdot w}) \vee (\neg x_{u \cdot w} \wedge \neg x_{v \cdot w})] \right) \; . \end{aligned}$$

Intuitively, the first part of ψ_{cl} requires that for a representative u there is a representative v such that their rows are equal ($y_{u,v} = 1$). The second part requires that the assignment of the values to the variables of the rows of u and v indeed are equal if $y_{u,v}$ is set to 1.

The following lemma shows that ψ_{cl} can be used to express the closed condition for an incomplete observation table.

Lemma 4.14. *There exists an assignment of Boolean values to the variables $x_{w_1}, \ldots, x_{w_n}$ such that the incomplete observation table is closed if and only if there is one to the variables $y_{u_1,v_1}, \ldots, y_{u_k,v_l}$, $k = 1, \ldots, |R \cdot \Sigma|$, $l = 1, \ldots, |R|$, with*

$$x_{w_1}, \ldots, x_{w_n}, y_{u_1,v_1}, \ldots, y_{u_k,v_l} \models \psi_{cl} \; .$$

Proof. Assume that there is an assignment of the variables $x_{w_1}, \dots, x_{w_n}$ such that the incomplete observation table is closed. Then, there is a representative $v \in R$ for every representative $u \in R \cdot \Sigma$ such that their rows are equal. We now set $y_{u,v} = 1$ if the rows of u and v are equal. Thus,

$$x_{w_1}, \dots, x_{w_n}, y_{u_1,v_1}, \dots, y_{u_k,v_l} \models \psi_{cl}$$

holds.

Now, assume that $x_{w_1}, \dots, x_{w_n}, y_{u_1,v_1}, \dots, y_{u_k,v_l} \models \psi_{cl}$ holds. Then, there is a $v \in R$ for every $u \in R \cdot \Sigma$ such that $y_{u,v} = 1$ because of the first part of ψ_{cl}. Moreover, from the last part of ψ_{cl} we know that

$$\bigwedge_{w \in S} [(x_{u \cdot w} \wedge x_{v \cdot w}) \vee (\neg x_{u \cdot w} \wedge \neg x_{v \cdot w})]$$

holds for all $y_{u,v}$ with $y_{u,v} = 1$. Thus, the rows of u and v are equal and $x_{w_1}, \dots, x_{w_n}$ is an assignment of the variables that makes the incomplete observation table closed.

□

The new formula ψ_{cl} can now easily be transformed into CNF. This is done in Figure 4.6. To avoid an exponential blow-up, we added auxiliary variables. This shifts the complexity from a high quantity of clauses to a slightly higher number of variables. Observation 4.15 states that both quantities are polynomial in the size of the incomplete observation table.

Observation 4.15. Let $O = (R, S, T)$ be an incomplete observation table. Then, the total number of variables of the formula ψ_{cl} in CNF are in

$$\mathcal{O}(|(R \cup R \cdot \Sigma) \cdot S| + |R \cdot \Sigma| \cdot |R|)$$

and the total number of clauses is in

$$\mathcal{O}(|R \cdot \Sigma| + |R \cdot \Sigma| \cdot |R| \cdot |S|) \ .$$

◁

Consistent formula

Again, let us briefly recall the meaning of the consistent condition. An observation table is consistent if for every two representatives $u \sim_O v \in R$ also $ua \sim_O va$ holds for all $a \in \Sigma$. In other words, if u and v have the same row, then for all $a \in \Sigma$ the representatives ua and va have to have the same row, too. This can be expressed formally by the following slightly more complicated first order formula.

$$\begin{aligned} \forall u, v \in R: \quad & (\forall w \in S: \ T(u \cdot w) = T(v \cdot w)) \\ & \to \ \forall a \in \Sigma \ \forall w \in S: \ T(u \cdot a \cdot w) = T(v \cdot a \cdot w) \end{aligned}$$

This formula can again be easily translated into a Boolean formula φ_{co} over the variables x_w.

$$\begin{aligned} \varphi_{co} \quad = \quad & \bigwedge_{u,v \in R} \Bigg(\bigwedge_{w \in S} \Big[(x_{u \cdot w} \wedge x_{v \cdot w}) \vee (\neg x_{u \cdot w} \wedge \neg x_{v \cdot w}) \Big] \\ & \to \bigwedge_{a \in \Sigma} \bigwedge_{w \in S} \Big[(x_{u \cdot a \cdot w} \wedge x_{v \cdot a \cdot w}) \vee (\neg x_{u \cdot a \cdot w} \wedge \neg x_{v \cdot a \cdot w}) \Big] \Bigg) \end{aligned}$$

$$\begin{aligned}
\psi_{cl} \quad = \quad & \bigwedge_{u \in R\cdot\Sigma} \bigvee_{v \in R} y_{u,v} \\
& \wedge \bigwedge_{u \in R\cdot\Sigma, v \in R} \left(y_{u,v} \rightarrow \bigwedge_{w \in S} [(x_{u\cdot w} \wedge x_{v\cdot w}) \vee (\neg x_{u\cdot w} \wedge \neg x_{v\cdot w})] \right) \\
\equiv \quad & \bigwedge_{u \in R\cdot\Sigma} \bigvee_{v \in R} y_{u,v} \\
& \wedge \bigwedge_{u \in R\cdot\Sigma, v \in R} \left(\neg y_{u,v} \vee \bigwedge_{w \in S} [(x_{u\cdot w} \wedge x_{v\cdot w}) \vee (\neg x_{u\cdot w} \wedge \neg x_{v\cdot w})] \right) \\
\equiv \quad & \bigwedge_{u \in R\cdot\Sigma} \bigvee_{v \in R} y_{u,v} \\
& \wedge \bigwedge_{u \in R\cdot\Sigma, v \in R} \bigwedge_{w \in S} (\neg y_{u,v} \vee (x_{u\cdot w} \wedge x_{v\cdot w}) \vee (\neg x_{u\cdot w} \wedge \neg x_{v\cdot w})) \\
\equiv \quad & \bigwedge_{u \in R\cdot\Sigma} \bigvee_{v \in R} y_{u,v} \\
& \wedge \bigwedge_{u \in R\cdot\Sigma, v \in R} \bigwedge_{w \in S} ((\neg y_{u,v} \vee \neg x_{u\cdot w} \vee x_{v\cdot w}) \wedge (\neg y_{u,v} \vee x_{u\cdot w} \vee \neg x_{v\cdot w}))
\end{aligned}$$

Figure 4.6: The formula ψ_{cl} in CNF

As for the closed formula φ_{cl}, we use Boolean auxiliary variables $z_{u,v}$ and $a_{u,v,w}$ for $u, v \in R$, $w \in S$ to avoid an exponential blow-up when transformed into CNF. The meaning of a variable $z_{u,v}$ is

> "The auxiliary variable $z_{u,v}$ is set to 1 if and only if the rows of u and v are not equal"

while the meaning of a variable $a_{u,v,w}$ is

> "The auxiliary variable $a_{u,v,w}$ is set to 1 if and only if the rows of u and v are distinct by the column labeled with w".

We now rewrite the formula φ_{co} and gain

$$\begin{aligned}
\psi_{co} \quad = \quad & \bigwedge_{u,v \in R} \left(\neg z_{u,v} \rightarrow \bigwedge_{a \in \Sigma} \bigwedge_{w \in S} [(x_{u\cdot a\cdot w} \vee x_{v\cdot a\cdot w}) \wedge (\neg x_{u\cdot a\cdot w} \vee \neg x_{v\cdot a\cdot w})] \right) \\
& \wedge \bigwedge_{u,v \in R} \left(z_{u,v} \rightarrow \bigvee_{w \in S} a_{u,v,w} \right) \\
& \wedge \bigwedge_{u,v \in R} \bigwedge_{w \in S} \left(a_{u,v,w} \rightarrow [(x_{u\cdot w} \vee \neg x_{v\cdot w}) \wedge (\neg x_{u\cdot w} \vee x_{v\cdot w})] \right) .
\end{aligned}$$

Intuitively, the first part requires for all $a \in \Sigma$ the two rows of $u \cdot a$ and $v \cdot a$ to be equal if the rows of u and v are equal ($z_{u,v} = 0$). The second part of the formula expresses that, if the rows of u and v are not equal ($z_{u,v} = 1$), then there has to be a column $w \in S$ that districts both rows ($a_{u,v,w} = 1$). Finally, the last part requires the assignment of Boolean values to $x_{u\cdot w}$ and $x_{v\cdot w}$ to be different if $a_{u,v,w} = 1$ holds.

The following lemma proves that ψ_{co} can be used to express the consistent condition for an incomplete observation table.

Lemma 4.16. *There exists an assignment of Boolean values to the variables $x_{w_1}, \ldots, x_{w_n}$ such that the incomplete observation table is consistent if and only if there is one to the variables $z_{u_1,v_1}, \ldots, z_{u_k,v_l}$ and $a_{u_1,v_1,a_1}, \ldots, a_{u_k,v_l,a_m}$, $k = 1, \ldots, |R|$, $l = 1, \ldots, |R|$, $m = 1, \ldots, |S|$, with*

$$x_{w_1}, \ldots, x_{w_n}, z_{u_1,v_1}, \ldots, z_{u_k,v_l}, a_{u_1,v_1,a_1}, \ldots, a_{u_k,v_l,a_m} \models \psi_{co} \; .$$

Proof. Assume that there is an assignment of Boolean values to the variables $x_1, \ldots, x_n$ such that the incomplete observation table is consistent. We assign $z_{u,v} = 1$ if and only if the row of u and v are not equal. Moreover, we assign $a_{u,v,w} = 1$ if and only if w is the witness that the rows of u and v are not equal, i.e. $T(u \cdot w) \neq T(v \cdot w)$. With this assignment

$$x_{w_1}, \ldots, x_{w_n}, z_{u_1,v_1}, \ldots, z_{u_k,v_l}, a_{u_1,v_1,a_1}, \ldots, a_{u_k,v_l,a_m} \models \psi_{co}$$

holds.

Now, assume that $x_{w_1}, \ldots, x_{w_n}, z_{u_1,v_1}, \ldots, z_{u_k,v_l}, a_{u_1,v_1,a_1}, \ldots, a_{u_k,v_l,a_m} \models \psi_{co}$ holds. We show that for every pair of representatives $u, v \in R$ the consistent condition is fulfilled. From the first part of the formula we know that for all $u, v \in R$ either $z_{u,v} = 1$ or $\bigwedge_{a\in\Sigma} \bigwedge_{w\in S} [(x_{u\cdot a\cdot w} \vee x_{v\cdot a\cdot w}) \wedge (\neg x_{u\cdot a\cdot w} \vee \neg x_{v\cdot a\cdot w})]$ holds.

- If $z_{u,v} = 1$, then there is a $w \in S$ with $a_{u,v,w} = 1$ because of the second part of ψ_{co}. From the last part of the formula we deduce that then $x_{u\cdot v} = 1 \Leftrightarrow x_{v\cdot w} = 0$ holds. Thus, the rows of u and v are not equal and the consistent condition holds for the pair u, v.

- If $z_{u,v} = 0$, then $\bigwedge_{a\in\Sigma} \bigwedge_{w\in S} [(x_{u\cdot a\cdot w} \vee x_{v\cdot a\cdot w}) \wedge (\neg x_{u\cdot a\cdot w} \vee \neg x_{v\cdot a\cdot w})]$ is satisfied. Thus, the rows of ua and va are equal for every $a \in \Sigma$ and the consistent condition is satisfied for the pair u, v, too.

All in all, the consistent condition is satisfied for every pair of representatives of R and, therefore, the assignment of Boolean values to the variables $x_{w_1}, \ldots, x_{w_n}$ produces a consistent incomplete observation table.

□

We can now transform the formula ψ_{co} into CNF as shown in Figure 4.7. Again, we avoid an exponential blow-up of the number of clauses. The following observation states that both the total number of clauses and variables is polynomial in the size of the incomplete observation table.

Observation 4.17. Let $O = (R, S, T)$ be an incomplete observation table. Then, the total number of variables of the formula ψ_{co} in CNF is in

$$\mathcal{O}(|(R \cup R \cdot \Sigma) \cdot S| + |R|^2 + |R|^2 \cdot |S|)$$

$$
\begin{aligned}
\psi_{co} \quad = \quad & \bigwedge_{u,v \in R} \left(\neg z_{u,v} \to \bigwedge_{a \in \Sigma} \bigwedge_{w \in S} [(x_{u \cdot a \cdot w} \vee x_{v \cdot a \cdot w}) \wedge (\neg x_{u \cdot a \cdot w} \vee \neg x_{v \cdot a \cdot w})] \right) \\
& \wedge \bigwedge_{u,v \in R} \left(z_{u,v} \to \bigvee_{w \in S} a_{u,v,w} \right) \\
& \wedge \bigwedge_{u,v \in R} \bigwedge_{w \in S} \left(a_{u,v,w} \to [(x_{u \cdot w} \vee \neg x_{v \cdot w}) \wedge (\neg x_{u \cdot w} \vee x_{v \cdot w})] \right) \\
\equiv \quad & \bigwedge_{u,v \in R} \left(z_{u,v} \vee \bigwedge_{a \in \Sigma} \bigwedge_{w \in S} [(x_{u \cdot a \cdot w} \vee x_{v \cdot a \cdot w}) \wedge (\neg x_{u \cdot a \cdot w} \vee \neg x_{v \cdot a \cdot w})] \right) \\
& \wedge \bigwedge_{u,v \in R} \left(\neg z_{u,v} \vee \bigvee_{w \in S} a_{u,v,w} \right) \\
& \wedge \bigwedge_{u,v \in R} \bigwedge_{w \in S} \left(\neg a_{u,v,w} \vee [(x_{u \cdot w} \vee \neg x_{v \cdot w}) \wedge (\neg x_{u \cdot w} \vee x_{v \cdot w})] \right) \\
\equiv \quad & \bigwedge_{u,v \in R} \bigwedge_{a \in \Sigma} \bigwedge_{w \in S} \left((z_{u,v} \vee x_{u \cdot a \cdot w} \vee \neg x_{v \cdot a \cdot w}) \wedge (z_{u,v} \vee \neg x_{u \cdot a \cdot w} \vee x_{v \cdot a \cdot w}) \right) \\
& \wedge \bigwedge_{u,v \in R} \left(\neg z_{u,v} \vee \bigvee_{w \in S} a_{u,v,w} \right) \\
& \wedge \bigwedge_{u,v \in R} \bigwedge_{w \in S} \left((\neg a_{u,v,w} \vee x_{u \cdot w} \vee x_{v \cdot w}) \wedge (\neg a_{u,v,w} \vee \neg x_{u \cdot w} \vee \neg x_{v \cdot w}) \right)
\end{aligned}
$$

Figure 4.7: The formula ψ_{co} in CNF

and the total number of clauses is in

$$\mathcal{O}(|R|^2 \cdot |\Sigma| \cdot |S| + |R|^2 + |R|^2 \cdot |S|) \; .$$

◁

The extended learner

The $L^*_?$ learner uses an incomplete observation table to store the learned information about the target language. It starts with an initial incomplete observation table $O = (R, S, T)$ with $R = S = \{\varepsilon\}$ and uses the function update(O) to ask membership queries for the unknown table entries. In the case that there are no ?-entries, the learner works similar to the original one. However, we have to adapt it a little bit to fit in the new context. As the original learner loops until the observation table is closed and consistent, it can now happen that a ?-value is added to the table during this loop. To handle this new situation, we do no longer use nested loops but only one main loop. In this main loop the learner decides whether it has to deal with ?-entries or not. In both cases, it checks successively whether the table is closed and consistent and finally constructs a conjecture. However, if the table contains ?-entries, we first have to compute an assignment of Boolean values to them. The extended learner is shown in pseudo code as Algorithm 4.

Let us first consider the case in which the incomplete observation table does not contain any ?-entries. In this case, the table is first checked to be closed and then to be consistent. If one of these conditions is not satisfied, we proceed as the original learner and add a word to R or S respectively. If the table is both closed and consistent, the learner constructs a conjecture, asks the MAT for equivalence and processes a counter-example if necessary. If the conjecture turns out to be equivalent, then the learner returns it and terminates.

The situation in which the table contains ?-entries is more difficult to handle. As in the previous case, the learner checks successively whether the table is closed and consistent. Thereto, we have to determine which Boolean values to assign to the ?-entries. The learner first tries to compute a model μ for the formula $\psi_{cl} \wedge \psi_{co}$ by using a SAT solver as subroutine. If the solver returns such a model, then μ codes an assignment that makes the table closed and consistent. The learner can then construct a conjecture $\mathcal{A}_{O_\mu}$ by using the values of μ for the ?-entries in the table. As in the previous case, the learner asks an equivalence query on this conjecture and processes a counter-example if necessary. If the conjecture is equivalent, the learner returns it and terminates.

However, there may be no assignment that makes the table closed and consistent and, thus, there is no model for the formula $\psi_{cl} \wedge \psi_{co}$. Then, the learner tries to find a model μ for the closed formula ψ_{co}. If the solver returns a model $\mu \models \psi_{co}$, we know that this assignment produces a consistent but no closed table. Thus, there is a word ua for $u \in R$ and $a \in \Sigma$ with $[\![ua]\!]_{O_\mu} \cap R = \emptyset$. As in the original learner, we add ua to R, update the table and continue with the next loop.

If the learner also cannot find a model for ψ_{co}, it tries to compute a model for the formula ψ_{cl}, which makes the table closed but not consistent. Again, in the case that the solver returns such a model, we proceed analogous to the original learner and add a word aw with $T_\mu(u \cdot a \cdot w) \neq T_\mu(v \cdot a \cdot w)$ to S, update O and continue with the next loop.

Finally, there may be no assignment that makes the table closed or consistent. In this case, it is not clear how to proceed further. One possibility is to add an arbitrary word (and its prefixes) to R or S. We choose to add a word aw to S such that $w \in S$, $a \in \Sigma$ and $aw \notin S$. However, note that this is a heuristic procedure.

A second version of the extended learner

Since solving the satisfiability problem for Boolean formulae is in general NP-complete, the runtime of the $L^*_?$ learner is dominated by the runtime of the SAT solver. Thus, one way to reduce the runtime and memory consumption is to reduce the number of variables and clauses generated.

Our approach to do this, which we call $L^*_{?\ V2}$ learner, is to replace all ?-entries in the incomplete observation table with the values of the recently computed model μ. Formally, we replace T with T_μ every time a new model μ is computed by the SAT solver. Note that, in contrast to the $L^*_{?\ V2}$ learner, the $L^*_?$ learner uses a computed model only temporarily in the current loop and discards it in the next.

Input: A MAT for a regular language with "don't cares"
Output: An equivalent DFA $\mathcal{A}$

```
Create an initial incomplete observation table O = (R, S, T);
repeat
    if O has ?-entries then
        if a model μ ⊨ ψ_cl ∧ ψ_co exists then
            Construct the conjecture A_{O_μ};
            if the MAT replies returns a counter-example w on an
            equivalence query then
                R := R ∪ pref(w);
                update(O);
            else
                return A_{O_μ};
            end
        else if a model μ ⊨ ψ_co exists then
            Choose a u ∈ R and an a ∈ Σ such that [[ua]]_{O_μ} ∩ R = ∅ ;
            R := R ∪ pref(w);
            update(O);
        else if a model μ ⊨ ψ_cl exists then
            Choose u, v ∈ R with u ~_{O_μ} v and w ∈ S, a ∈ Σ such that
            T_μ(uaw) ≠ T_μ(vaw) ;
            S := S ∪ {aw} ;
            update(O);
        else
            Use a heuristic method to extend the table;
        end
    else
        if O is not closed then
            Choose a u ∈ R and an a ∈ Σ such that [[ua]]_O ∩ R = ∅ ;
            R := R ∪ {ua} ;
            update(O) ;
        else if O is not consistent then
            Choose u, v ∈ R with u ~_O v and w ∈ S, a ∈ Σ such that
            T(uaw) ≠ T(vaw) ;
            S := S ∪ {aw} ;
            update(O) ;
        else
            Construct the conjecture A_O ;
            if the MAT returns a counter-example w on an equivalence
            query then
                R := R ∪ pref(w) ;
                update(O) ;
            else
                return A_O;
            end
        end
    end
until False ;
```

Algorithm 4: The $L^*_?$ learner

However, this approach has some weaknesses, which are mainly caused by the fact that we have no influence on the model. Moreover, the extended learner looses the ability to change its behavior on non XML word representations since the model now determines it once.

Correctness of the $L^*_?$ and $L^*_{?\ V2}$ learners

It is not hard to verify that the correctness of both learners follows immediately from their termination. However, because the results of a SAT solver are not predictable and the target language is not unique, it is very hard to deduce general facts about the $L^*_?$ and $L^*_{?\ V2}$ learners and in particular about their termination. Note that we do not address a formal proof of their termination in this thesis.

4.2.2 RPNI FOR REGULAR LANGUAGES WITH "DON'T CARES"

We now present our heuristic method to learn regular languages with "don't cares" called RPNI$_?$. This heuristic method relies on the RPNI algorithm, which infers a DFA from a finite sample $S = (S_+, S_-)$. In fact, it is the application of a non-query algorithm to the active learning of regular languages with "don't cares". However, the learner RPNI$_?$ does not change the original RPNI algorithm but uses it as a subroutine.

For a regular language with "don't cares" (L, L_r) the idea of this learner is to incrementally extend the sample S by either adding new words to S_+ or to S_-. Every time a new word is added the learner invokes the original RPNI algorithm to infer a conjecture $\mathcal{A}$ based on the new sample. This conjecture is used on an equivalence query. If the current conjecture is equivalent, then the learner returns it and terminates. If the conjecture is not equivalent, then the MAT returns a counter-example $w \in L(\mathcal{A}) \cap L_r \Leftrightarrow w \notin L$, which is then used to extend the sample:

- If $w \in L(\mathcal{A}) \cap L_r$, then the counter-example is accepted by the conjecture but it has to be rejected by every equivalent DFA since $w \notin L$. Hence, we ensure this by adding w to S_-.

- If $w \notin L(\mathcal{A}) \cap L_r$, then the counter-example is rejected by the conjecture. However, every equivalent DFA has to accept it. Thus, we ensure this by adding w to S_+.

The learner RPNI$_?$ can be found in pseudo code as Algorithm 5.

Before we give an example run of this learner, we briefly note that the learner is well defined. Thereto, recall that the RPNI algorithm requires S_+ to be nonempty. Furthermore, assume that $L \neq \emptyset$. Since the learner starts with a conjecture accepting the empty language, the first equivalence query provides a counter-example that is in L. Thus, this counter-example is added to S_+ and hence the RPNI algorithm runs always on a nonempty set of positive samples. In the case of $L = \emptyset$, the initial conjecture is already equivalent and thus the RPNI algorithm is never executed.

Let us now consider the following example to get a better understanding of the presented learner.

Input: A MAT for a regular language with "don't cares"
Output: An equivalent DFA $\mathcal{A}$

$S_+ = S_- = \emptyset$;
Let $\mathcal{A}$ be the trivial DFA with $L(\mathcal{A}) = \emptyset$;
while *the MAT replies with a counter-example to the equivalence query on the conjecture* $\mathcal{A}$ **do**
    Let $w \in L(\mathcal{A}) \cap L_r \Leftrightarrow w \notin L$ be a counter-example;
    if $w \in L(\mathcal{A})$ **then**
        $S_- := S_- \cup \{w\}$;
    else
        $S_+ := S_+ \cup \{w\}$;
    end
    $S := (S_+, S_-)$;
    $\mathcal{A} := RPNI(S)$;
end
return $\mathcal{A}$;

Algorithm 5: The RPNI$_?$ learner

Example 4.18. This example is intended to demonstrate two things. First, we want to show an example run of the RPNI$_?$ learner. Second, we want to illustrate how a learning algorithm for regular languages with "don't cares" can be used to learn a DFA for validating XML word representations. Therefore, consider the DTD

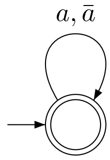
$$d = a \to a^?$$

where $\Sigma_{tag} = \{a\}$ and let $L = L(d)$ and $L_r = L^{XML}$.

The RPNI$_?$ learner starts with the trivial DFA $\mathcal{A}$ that accepts the empty language. Since $\mathcal{A}$ and d are obviously not equivalent, the MAT replies to the equivalence query with the conjecture $\mathcal{A}$ with a counter-example, say $w = a\bar{a} \in L(d)$. Since $w \notin L(\mathcal{A}) \cap L_r$, the algorithm adds w to S_+ and we gain

$$S_+ = \{a\bar{a}\} \text{ and } S_- = \emptyset \ .$$

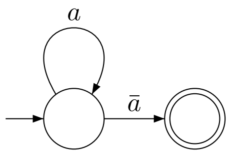
Now, the RPNI algorithms constructs a new conjecture for the sample $S = (S_+, S_-)$. This conjecture is shown in the following picture.



It is easy to see that this DFA is not equivalent to d because it accepts Σ^*. An equivalence query with this conjecture yields the counter-example, say $w = aa\bar{a}a\bar{a}\bar{a}$, which is added to S_-. We gain

$$S_+ = \{a\bar{a}\} \text{ and } S_- = \{aa\bar{a}a\bar{a}\bar{a}\} \ .$$

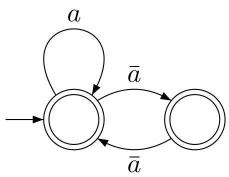
The RPNI algorithm is invoked and yields a new conjecture, which is shown below.



This conjecture is also not equivalent and the MAT responds with a counter-example $w = aa\bar{a}\bar{a}$, which is added to S_+ since $aa\bar{a}\bar{a} \notin L(\mathcal{A}) \cap L_r$. This leads to

$$S_+ = \{a\bar{a}, aa\bar{a}\bar{a}\} \text{ and } S_- = \{aa\bar{a}a\bar{a}\bar{a}\} .$$

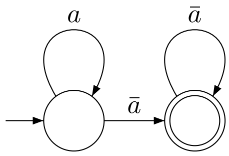
Again, the RPNI algorithm infers a new conjecture, which is depicted below.



Since this conjecture is still inequivalent to d, the MAT responds with a counter-example $w = aaa\bar{a}\bar{a}a\bar{a}\bar{a}$. The counter-example is added to S_- and the new sample is given by

$$S_+ = \{a\bar{a}, aa\bar{a}\bar{a}\} \text{ and } S_- = \{aa\bar{a}a\bar{a}\bar{a}, aaa\bar{a}\bar{a}a\bar{a}\bar{a}\} .$$

The RPNI algorithm constructs the conjecture depicted below.



This conjecture is finally equivalent to d and the RPNI$_?$ learner terminates and returns it.

◁

Correctness of the RPNI$_?$ learner

As for the $L^*_?$ and $L^*_{?\ V2}$ learners, the correctness of the RPNI$_?$ learner follows directly from its termination. However, in contrast to them, the RPNI$_?$ learner does not use an external SAT solver. We observe that this has a positive effect on the runtime and memory consumption. As for the learners based on Angluin's algorithm, it is very hard to show general facts about the *RPNI$_?$* learner and in particular about its termination. This is because it is based exclusively on equivalence queries, whose answers are not unique. This does, moreover, show that the RPNI$_?$ learner does not use the full information available. Note that we do not address a formal proof of its termination in this thesis.

4.2.3 Further improvements

Several improvements have been proposed to Angluin's learner since she published her paper in 1987. A good overview and comparison of several known variants was done by Balcázar, Díaz and Gavaldà in [BDG97]. Next, we introduce two of them and study their applicability and relationship to our setting. Unfortunately, it turns out that these improvements exploit fundamental properties of regular languages, which seems to make it impossible to adapt their function. Anyway, the analysis of these improvements may reveal useful insights.

A straightforward improvement is described by Alur, Madhusudan and Nam [AMN05] (also see [BDG97]). It realizes the idea that both the runtime of the learner and the number of membership and equivalence queries can be decreased significantly if the used observation table is as small as possible. To achieve this, exactly one representative for each equivalence class (or state of the conjecture) is stored. Of course, the information about $R \cdot \Sigma$ is also stored in the table. These kind of observation tables are called *reduced observation tables*. The representatives are called *access strings* since each representative determines uniquely a state of the conjecture reached after reading it. The use of a reduced observation table does not only reduce the number of representatives stored but also makes it redundant to check whether the table is consistent since the rows of every two representatives are distinct and, thus, the table is consistent by definition. Moreover, a closing operation preserves the uniqueness of the representative's rows as a representative with a new distinct row is added. The construction of a conjecture is now easy and uses each representative as a state of the DFA.

This more compact observation table makes it necessary to handle counter-examples differently since we have to ensure that each representative has a unique row. Instead of adding all prefixes of a returned counter-example to the table, the counter-example is analyzed in order to find the longest suffix that witnesses the difference between the language accepted by the conjecture and the target language. Intuitively, the conjecture guessed wrong since this point because the representative of the reached state and the prefix of the counter-example are not language-equivalent. After identifying such a suffix, it is added as new sample. We argue in a moment that after this operation the observation table is no longer closed.

In the analysis itself the run of the conjecture on the counter-example uav is simulated. The goal is to identify the first situation

$$\ldots \xrightarrow{u} r \xrightarrow{a} r' \xrightarrow{v} \ldots$$

in the run such that $r'v \in L \Leftrightarrow uav \notin L$ by means of membership queries. Such a situation must exist since $r'' \cdot \varepsilon \in L \Leftrightarrow uav \cdot \varepsilon \notin L$ holds for the representative r'' reached after reading the whole counter-example uav. In this case, we know that the equivalence class of r' has to be split up and we add v as new sample. It is not hard to verify that after this operation the table is no longer closed since the v-entry of the rows of ra and r' differ and, thus, r' is no longer the representative of the a-successor of r. A slightly improved version uses a binary search to identify the suffix added to the table instead of a linear search as described.

In total, the algorithm presented uses at most $\mathcal{O}(|\Sigma| \cdot n^2 + n \cdot \log m)$ membership and n equivalence queries, where n is the number of states of the canonical

DFA $\mathcal{A}_L$ and m is the length of the longest counter-example. Since all performed operations can be done efficiently, the algorithm has a polynomial runtime in n and m. A detailed runtime analysis can be found in [BDG97].

Kearns and Vazirani [KV94] presented an even more sophisticated variant of Angluin's algorithm. They use a binary *classification tree* as data structure to maintain representatives and samples. Each leave node is labeled with a representative $u \in R$ while each internal node is labeled by a sample $w \in S$. The tree is constructed by choosing a root node $w \in S$ that distinguishes two representatives and subsequently partitioning all representatives of R. The representatives $u \in R$ with $u \cdot w \in L$ are put in the right subtree and the representatives with $u \cdot w \notin L$ are put in the left subtree. Now, we repeat this at each subtree, choosing a new sample until each representative has its own leaf node. The algorithm has to ensure that the root of the classification tree is always labeled with ε and that ε is also a representative. Note that, in contrast to a two dimensional observation table, a classification tree only uses a subset of the samples to distinguish two representatives.

A classification tree induces a partition over the words of Σ^*. The block of a word $v \in \Sigma^*$ can be computed by *sifting v down* the classification tree using membership queries. Starting with the root node, at each node labeled with the sample w we make a membership query on $v \cdot w$ and branch left or right depending on the answer of the MAT (left if the answer is $v \cdot w \notin L$ or right otherwise) until a leaf node is eventually reached. In fact, this partition naturally induces an equivalence relation where two words u and v are equivalent if they are sifted down to the same leaf node. Particularly, if $u \sim_L v$, then u and v are obviously equivalent w.r.t. any classification tree. Moreover, we use the labels of the leaf nodes as representatives of the equivalence classes.

With this notation, we can now describe the construction of a conjecture DFA. Each representative, i.e. each leave node, defines a state of the conjecture and for each representative u and each input symbol $a \in \Sigma$ the destination of the a-transition is defined as the representative that is reached after sifting ua down the classification tree.

We now make an equivalence query on this conjecture. In the case that a counter-example uav is returned, we analyze it in a similar manner as for reduced observation tables and identify the longest suffix that witnesses the difference between the language accepted by the conjecture and the target language. Thereby, we have to compute the representative r of a state reached after reading a prefix ua of the counter-example and the representative r' of the equivalence class of ua (computed by sifting ua down the classification tree). The goal is to identify the first situation where $r \neq r'$. Let us assume that this happens the first time after reading ua. Then, we know that the equivalence class of u has to be split up since this equivalence class contains two words with different a-successors. To do so, we update the classification tree and replace the leaf node labeled with the representative r reached by sifting u down the classification tree with an internal node with two leaf nodes. One leaf node is labeled with r and the other with the new representative u. The internal node is labeled with the new sample ad where d is the unique distinguishing sample for r and u, which can be obtained from the classification tree.

Note that the Kearns-Vazirani algorithm builds a conjecture every time from scratch. However, it is also possible to store the needed information in an auxiliary data structure and only compute the information needed after updating the classification tree. Thus, this algorithm makes $\mathcal{O}(|\Sigma| \cdot n^2 + n \cdot \log m)$ membership and at most n equivalence queries. Since all performed operations, especially all sifting operations and the update of the classification tree, can be done efficiently, the algorithm has a runtime polynomial in n and m. Again, a detailed runtime analysis can be found in [BDG97].

An adaptation of one of these variants to the setting of learning regular languages with "don't cares", particularly the algorithm described by Alur, Madhusudan and Nam, could result in a better performance. Therefore, in the following we discuss the possibilities and problems of an adaptation.

The use of a reduced observation table and, especially, the more efficient way of handling counter-examples leads to a much smaller incomplete observation table and SAT formulae. However, both improvements exploit the fact that the equivalence classes of a regular language can be exactly determined. However, this does not apply to our setting. The target language is not unique and, hence, we do not know which equivalence classes we shall learn. To illustrate this, we assume that a state r of a conjecture is reached after reading a prefix u of a returned counter-example uv. Moreover, let us assume that the MAT returns "?" to an equivalence query on rv. In this situation it is not clear which value to assign to the ?-value and, thus, whether to split the current equivalence class up or to test the next one. Furthermore, if we add v as new sample, it can surely happen that the solver computes a value for this ?-value in one of the next steps, which is different to the one we have currently chosen. In the worst case, this can lead to a situation where the same conjecture is constructed repeatedly and, hence, the same counter-example is returned.

The binary classification tree used in the Kearns-Vazirani algorithm relies on the fact that the result of a membership query during a sifting operation can be either "yes" or "no" but a "don't care" answer cannot be handled. This would require extending the tree in some way. One possibility is to use a tertiary tree where one branch of an internal node is used to handle "don't care" answers while the other two are used in the same way as in the original algorithm. However, it is unclear how we can transform such a tree reasonably into a binary tree to construct a conjecture DFA.

The described problems prevent us from easily adapting either the use of a reduced observation table or a Kearns-Vazirani like algorithm.

4.2.4 Valid pair improvement

A direct adaptation of improvements to Angluin's learner seems not to work for the "don't care" case. Hence, we reconsider the validation of XML word representations and describe an improvement for this special case. Recall that all presented learning algorithms are general methods for learning regular languages with "don't cares". Therefore, they do not exploit any special structural properties of DTDs. The idea of our improvement is to add more information about the given DTD to the learners.

Each XML word representation is a sequence $w = a_1 \ldots a_n$ of opening and closing tags from $\Sigma = \Sigma_{tag} \cup \bar{\Sigma}_{tag}$. We observe that, if $w \in L(d)$ for a given DTD $d = (\Sigma_{tag}, r, \rho)$, then not every arbitrary pair of consecutive tags $a_i a_{i+1}$ is allowed to occur in w. In fact, each individual rule of the DTD determines which pairs can occur in a valid XML word representation. The next example makes this observation clear.

Example 4.19. Consider the DTD

$$d = \begin{array}{l} r \to a \\ a \to r^? \end{array} .$$

The pairs ra, ar, $a\bar{a}$, $\bar{r}\bar{a}$ and $\bar{a}\bar{r}$ are valid since they can occur in a valid XML word representation. The pairs rr, $r\bar{r}$, $r\bar{a}$, aa, $a\bar{r}$, $\bar{r}r$, $\bar{r}a$, $\bar{r}\bar{r}$, $\bar{a}r$, $\bar{a}a$ and $\bar{a}\bar{a}$ are not valid.

◁

The valid pair improvement exploits this property. The basic idea is to force that, during the learning process, a conjecture does only accept words in which every two consecutive symbols are *valid pairs*. To formalize our intuition, we first define valid pairs.

Definition 4.20. [Valid pair] Let $d = (\Sigma_{tag}, r, \rho)$ be a DTD and $\Sigma = \Sigma_{tag} \cup \bar{\Sigma}_{tag}$. The pair $(a, b) \in \Sigma \times \Sigma$ is called a *valid pair* if and only if there is a $w \in L(d)$ and $u, v \in \Sigma^*$ such that

$$w = uabv .$$

The set of all valid pairs is denoted by VP_d.

◁

Since every word in $L(d)$ starts with r and ends with $\bar{r}$, we call $w = a_1 \dots a_n$ where $a_1 = r$, $a_n = \bar{r}$ and $(a_i, a_{i+1}) \in VP_d$ for $i = 1, \dots, n-1$ a *valid tag sequence*. Note that $L(d) \subseteq \{w \in \Sigma^* \mid w \text{ is a valid tag sequence}\}$.

To understand how to compute the set of valid pairs, we consider a Σ_{tag}-valued tree t that is valid w.r.t. d. We observe that a valid pair can occur as one of the following four types:

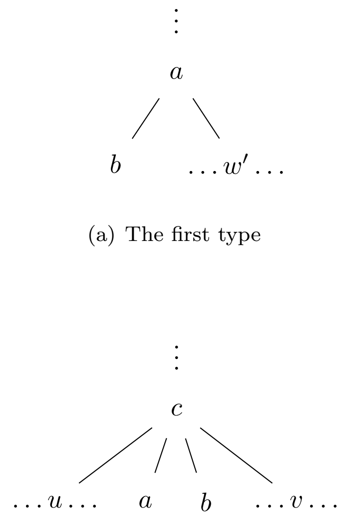
1. A valid pair consists of two consecutive opening tags (a, b). This situation occurs if a word $w \in L(\rho(a))$ begins with b, i.e. $w = bw'$. In other words the first child of an a labeled node of t is labeled with b. This situation is shown in Figure 4.8(a).

2. A valid pair consists of an opening a tag followed by a closing tag $\bar{a}$. This situation can only occur if $\varepsilon \in L(\rho(a))$, i.e. an a labeled node of t is a leaf node. This situation is depicted in Figure 4.8(b).

3. A valid pair consists of a closing tag $\bar{a}$ followed by an opening tag b. This situation occurs if there is a tag c and a word $w \in L(\rho(c))$ such that $w = uabv$ for some $u, v \in \Sigma_{tag}^*$. This situation is shown in Figure 4.8(c).

4. A valid pair consists of two consecutive closing tags $(\bar{a}, \bar{b})$. This is the converse situation to the first one. It occurs if a word $w \in L(\rho(b))$ ends with a, i.e. $w = w'a$. This situation is depicted in Figure 4.8(d).

We can check for these situations by individually examining each tag $a \in \Sigma_{tag}$ and their associated horizontal language. However, it is more reasonable to use the horizontal DFA $\mathcal{A}_a = (Q_a, \Sigma_{tag}, q_0^a, \delta_a, F_a)$ than the regular expression $\rho(a)$. Thereby, we assume that Q_a only contains *useful* states. A state is called



(a) The first type (b) The second type

(c) The third type (d) The fourth type

Figure 4.8: Types of valid pairs illustrated on a part of a Σ_{tag}-valued tree t

useful if it is reachable from the initial state and if a final state can be reached. This ensures that a DFA does not contain states that are never used.

Let us now describe how the valid pairs of each type can be computed.

1. Consider valid pairs of the first type. We can identify these pairs by looking at the initial states of the horizontal DFAs $\mathcal{A}_a$. Thereto, consider all outgoing transitions from the initial state q_0^a. If there is an outgoing transition labeled with b, we know that (a, b) is a valid pair.

2. Now, consider the valid pairs of the second type. Since the pair $(a, \bar{a})$ is valid if $\varepsilon \in L(\rho(a)) = L(\mathcal{A}_a)$, we mark all pairs $(a, \bar{a})$ as valid if q_0^a is a final state of $\mathcal{A}_a$.

3. Consider the third type of valid pairs. To compute valid pairs of this type, we have to examine every state q of a horizontal DFA $\mathcal{A}_c$ and compute all incoming transitions $\delta_c(q', a) = q$ as well as all outgoing transitions $\delta_c(q, b) = q''$. We then mark the pair $(\bar{a}, b)$ as valid.

4. Finally, consider the fourth type of valid pairs. We can identify such pairs by considering the incoming a-labeled transitions to final states of $\mathcal{A}_b$. Every time we discover such a situation we mark the pair $(\bar{a}, \bar{b})$ valid.

An algorithm that computes the set VP_d of valid pairs for a DTD d is shown in Algorithm 6.

It is not hard to verify that the set of all valid tag sequences of a DTD is a regular language. A DFA that stores the last read symbol of the input in its states can easily accept this language. With this information and the knowledge of all valid pairs it is able to decide whether the pair consisting of last symbol

Input: The DFAs $\mathcal{A}_a = (Q_a, \Sigma_{tag}, q_0^a, \delta_a, F_a)$ for all $a \in \Sigma_{tag}$
Output: The set VP_d

```
VP_d := ∅;
forall a ∈ Σ_tag do
    forall states q ∈ Q_a do
        Let S = {c̄ ∈ Σ̄_tag | ∃q' ∈ Q_a : δ_a(q', c) = q} be the set of all
        incoming c transitions to the state q;
        if q = q_0^a then
            S := S ∪ {a};
        end
        if q ∈ F_a then
            VP_d := VP_d ∪ S × {ā};
        end
        forall c ∈ Σ_tag do
            VP_d := VP_d ∪ S × {c};
        end
    end
end
return VP_d;
```

Algorithm 6: Computing the set of all valid pairs VP_d for a DTD d

and the current symbol in the input is a valid pair or not. In the first case, it switches its control state to remember the current input symbol. In the latter case, it switches to a sink state. Since every valid tag sequence starts with the root symbol, we require that first symbol of the input is r. Moreover, we require that every accepting run ends in the state representing the symbol $\bar{r}$. This ensures that the last symbol of the input is $\bar{r}$. We denote this informally described DFA as $\mathcal{A}_{VP_d}$. It is defined as shown in Construction 4.21.

Construction 4.21. *The DFA $\mathcal{A}_{VP_d} = (Q, \Sigma, q_0, \delta, F)$ is defined as*

- $Q = \Sigma \cup \{q_0, q_r\}$,
- $F = \{\bar{r}\}$ *and*
- δ *defined by*

$$\begin{aligned}
\delta(q_0, r) &= r, \\
\delta(q_0, b) &= q_r \quad \textit{for all } b \in \Sigma \setminus \{r\}, \\
\delta(q_r, b) &= q_r \quad \textit{for all } b \in \Sigma \\
\delta(a, b) &= \begin{cases} b & , \textit{if } (a, b) \in VP_d \\ q_r & , \textit{else} \end{cases}
\end{aligned}$$

The valid pair improvement performs the following two steps every time the learner asks an equivalence query.

1. First, the product $\mathcal{A}_{VP_d} \otimes \mathcal{A}$ of $\mathcal{A}_{VP_d}$ with the conjecture $\mathcal{A}$ provided by the learner is constructed.
2. Then, an equivalence query on $\mathcal{A}_{VP_d} \otimes \mathcal{A}$ is asked.

Note that the DFA $\mathcal{A}_{VP_d}$ only needs to be constructed once in the beginning of the learning process.

As a final remark, we observe that the information about valid pairs of a DTD is not used directly in the learning process. Thus, the learner itself is not changed but only the conjecture provided by it. This has the advantage that the learner does no longer need to learn the valid pairs from counter-examples and, thus, the counter-examples provided by the MAT can be considered to be more "useful". Moreover, the valid pair improvement can easily be applied to all learners for regular languages with "don't cares". However, a direct integration of structural information into the learning process may gain a much higher benefit.

4.2.5 Application to the validation of XML word representations

A learner for regular languages with "don't cares" can easily be applied to learn DFAs for the validation of XML word representations. For a recognizable DTD $d = (\Sigma_{tag}, r, \rho)$, the relevant inputs are XML word representations and "don't cares" correspond to non well-formed inputs, i.e. $L_r = L^{XML}$ and $L_d = \Sigma_{tag}^* \setminus L^{XML}$. Provided with a MAT for a recognizable DTD, a learner for regular languages with "don't cares" results in a DFA $\mathcal{A}$ with

$$L(d) = L^{XML} \cap L(\mathcal{A}) .$$

Remark 4.22. Since the termination of the presented learning algorithms for regular languages with "don't cares" was not addressed, we can also make no statement about the termination and, hence, the runtime of our learning methods for DFAs that validate XML word representations.

$\triangleleft$

4.3 A proof-of-concept implementation

This section presents the practical part of this thesis. Since the algorithms presented in Section 4.2 are heuristic methods, we decided to implement a proof-of-concept rather than addressing a formal proof of correctness. This proof-of-concept is a Java based software that allows to learn DFAs that validate XML word representations. The software consists of our learning algorithms for regular languages with "don't cares" and our MAT for XML word representations. Our intention is the following.

- On the one hand, we want to empirically show that our learning method for DFAs that validate XML word representations works. Moreover, we want to provide DFAs validating sample DTDs that cannot be addressed by the direct construction methods.

- On the other hand, we hope that a detailed analysis of the heuristics results in deeper understanding of their function.

This section starts with a description of our implemented code and finishes with a detailed analysis.

4.3.1 Implementation details

Our implementation is written in the Java 6 SE programming language.[3] Since it uses so called generics, it requires at least the Java 5 runtime environment. However, it is independent of the used operating system and has been tested with the Java 6 runtime environment on the Windows and Linux operating system.

Both the $L^*_?$ learner and the $L^*_{?\ V2}$ learner use a SAT solver as a subroutine. In our implementation, the SAT4J 1.7 RC 2 library written by Daniel Le Berre is used [Le 06]. This library provides a wide range of SAT solvers for small and large scale SAT instances. However, we use its default solver, which is best fitted to solve large SAT instances. The SAT4J library can be downloaded as a jar-file from its project website `http://sat4j.org/`. Some benchmarks can be found on the SAT4J project website and on the 2006 SAT-Race website `http://fmv.jku.at/sat-race-2006/`.

A solver of the SAT4J library requires the input to be a SAT instance in CNF. Internally a CNF formula is stored as a set of clauses. This allows an easy handling of a formula by simply adding or removing clauses. A model computed by a solver satisfies the conjunction of all clauses in this set.

The SAT4J library is completely written in Java and, thus, can be easily integrated into our code. However, the internal representation of variables differs: Our code uses `Strings` to identify variables while the SAT4J library simply enumerates them. To be precise, the SAT4J library represents a literal by an `int`-value. This value is positive if the literal is a variable and negative if it is the negation of a variable. Each solver of the SAT4J library stores a number $n > 0$ to indicate that clauses can be build with the literals $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$. A clause is represented by an array of `int`-values. E.g. the clause $x_1 \vee \neg x_2 \vee \neg x_3$ is transformed into `[1, -2, -3]`. Such `int`-arrays can then be added to a solver.

To generate the formulae ψ_{cl} and ψ_{co} it is, thus, necessary to map each variable x_w, $y_{u,v}$, $z_{u,v}$ and $a_{u,v,w}$ to a variable x_i. We do this by defining bijective functions

$$\begin{aligned} \nu_x &: (R \cup R \cdot \Sigma) \cdot S \to \mathbb{N}_+ \\ \nu_y &: R \cdot \Sigma \times R \to \mathbb{N}_+ \\ \nu_z &: R \times R \to \mathbb{N}_+ \\ \nu_a &: R \times R \times S \to \mathbb{N}_+ \end{aligned}$$

that assign a unique number to each `String`-indexed variable. These functions are realized as nested hash-maps.

We now present a more detailed view at our code. The Java code is organized in several packages, which we now discuss. For detailed information about each class or method, we refer to the Java-Doc contained in the source code or the HTML version of the Java-Doc.

The angluin-package

The `angluin`-package contains the implementation of both the $L^*_?$ and the $L^*_{?\ V2}$ learner.

[3]For more information see `http://java.sun.com/`

The class `ObservationTable` realizes an incomplete observation table. The sets R and S are stored as hash-sets while the mapping T is stored as a hash-map. An `ObservationTable`-type object covers all necessary operations and provides methods like

- extending R and S while the prefix- and suffix-closeness is preserved as well as updating the mapping T,
- replacing and returning values of the mapping T,
- checking the table to be closed or consistent in both cases with and without ?-entries,
- constructing the formulae ψ_{cl} and ψ_{co},
- computing models for the Boolean formulae and
- constructing a conjecture in both cases with and without ?-entries.

To construct the formulae ψ_{cl} and ψ_{co}, each clause is processed individually. Thereby, it is first checked whether a clause does not evaluate to `true` if the variables x_w of all occurring non ?-entries are substituted with the values $T(w)$. If a clause evaluates to `true` because of this substitution, the remaining variables of this clause can have arbitrary values. Hence, we do not add this clause to the SAT solver. By contrast, if a clause is not evaluated to `true` and, thus, is needed to determine the value of the variables in this clause, it is added to the solver. Therefore, it is possible to reduce the number of clauses and auxiliary variables.

Since the solver uses `int`-values to identify variables, we use the functions ν_x, ν_y, ν_z and ν_a to assign a unique number to the variables x_w, $y_{u,v}$, $z_{u,v}$ and $a_{u,v,w}$. For better reading, we represent those functions by a single function ν. Then, a clause $\bigvee_{i=1}^{n} l_i$, where l_i is a literal for all $i \in \{1, \dots, n\}$, is transformed into the `int`-vector $[\nu(l_1), \dots, \nu(l_n)]$ before it is added to the solver.

To construct ψ_{cl} and ψ_{co}, a special method is each provided. These methods are designed to be executed in sequence such that $\psi_{cl} \wedge \psi_{co}$ can easily be constructed by first executing the ψ_{cl}-method and then the ψ_{co}-method. Each generates clauses and directly adds them to the solver.

Finally, the SAT solver can be invoked to compute a model μ. If a model is found, then the inverse mapping $\nu_x^{-1} : \mathbb{N}_+ \to (R \cup R \cdot \Sigma) \cdot S$ is used to extract the values of the computed model. If μ is a model for the formula $\psi_{cl} \wedge \psi_{co}$, it can be used to construct a conjecture.

The class `LearningAlgorithmAngluin` is the implementation of Algorithm 4. It realizes both the $L^*_?$ and the $L^*_{?\ V2}$ learner in two different methods. The return value of these methods is the learned DFA as an `FSA`-type object.

The automaton-package

The `automaton`-package contains the implementations of the automata used by the learning algorithms. In particular, this package contains implementations of the following automata models.

- The class FSA provides an implementation of a deterministic finite automaton.

- The class PAutomaton is an implementation of a P-automaton used by the saturation algorithm. In contrast to objects of type FSA, a PAutomaton-type object realizes a nondeterministic finite automaton.

- The class PDA provides an implementation of a deterministic pushdown automaton. It is used by the MAT for XML word representations. Note that this class does not implement a visibly pushdown automaton as introduced in Section 2.3 but a deterministic pushdown automaton with ε-transitions. This automaton model allows no actions on the empty stack and accepts if the input is completely processed and a final state is reached.

Additionally, this package contains several auxiliary classes, which are only of minor interest.

The exceptions-package

All exceptions, which can be thrown during a learning process, are bundled in the exceptions-package.

The main-package

The main-package contains classes for general purposes.

The classes SimpleAlphabet and TagAlphabet are implementations of alphabets. The class SimpleAlphabet realizes a simple set of symbols. The class TagAlphabet represents an alphabet of tags where each opening tag is associated with the corresponding closing tag and vice versa. Moreover, this class ensures that the set of opening and closing tags are disjoint. Both the class SimpleAlphabet and the class TagAlphabet implement the interface Alphabet, which defines the minimum requirements of an alphabet and allows a generic use. In all cases, the symbols have the type char.

The main-package contains the class DTD, an implementation of a DTD. A DTD-type object stores the TagAlphabet used, the root symbol and a horizontal language DFA for every tag of the alphabet. Moreover, the class DTD provides a method to construct the automaton $\mathcal{A}_d$ for this DTD. This method also returns the associated P-automaton as defined in Section 2.3. Moreover, the class DTD provides a method to construct the DFA $\mathcal{A}_{VP_d}$.

The class DTDFactory provides predefined DTDs, which we use for experiments and the analysis of the learners. These DTDs are shown in Figure 4.9.

The saturation algorithm is realized in the class SaturationAlgorithm. This class provides several implementations of the saturation algorithm. All of them rely on the algorithm proposed by Bouajjani, Esparza and Maler [EHRS00]. However, only the extended saturation algorithm is capable of computing a counter-example directly. As mentioned earlier, this counter-example may not be the canonically smallest one.

To compute a smallest counter-example, the `main`-package provides the class `MinimalCounterExampleGenerator`. This class computes all reachable configurations from a given set of configurations with increasing distance. It terminates if a given configurations is reached. However, to ensure that this procedure eventually terminates, the saturation algorithm has to be invoked prior.

A user interface is realized as a command line interface in the class `Learner`. It is used to invoke the learning algorithms customizable by command line options. Moreover, it can be used to display the predefined DTDs.

The mat-package

The MAT of Section 4.1 is realized in the `mat`-package. A MAT implements the interface `Oracle`. Like this it is possible to use an arbitrary MAT in a learning process.

The class `FSAOracle` implements a MAT for regular languages. It uses an object of type `FSA` as representation of the target language. We used this class for testing purposes during the programming.

The class `DTDOracle` implements a MAT for XML word representations. It uses the automaton $\mathcal{A}_d$ to answer membership queries and the saturation algorithm to answer equivalence queries. Moreover, it is possible to compute a smallest counter-example, which is done by the `MinimalCounterExampleGenerator` in the `main`-package.

The rpni-package

The RPNI$_?$ learner is located in the `rpni`-package. The learning algorithm itself is realized in the class `LerningAlgorithmRPNI`. This class uses objects of the type `EquivalenceClass` to represent equivalence classes and an object of the type `Congruence` to represent a set of equivalence classes that forms a congruence relation. A congruence relation stores equivalence classes as linked lists to perform merging operations efficiently.

The class `CanonicalStringComparator` replaces the standard way of comparing `Strings` in Java, which is the lexicographical order.

The test-package

The `test`-package provides JUnit test cases for all major classes of this software distribution. Each class in this package represents a test suite for a specific class under test and contains several test cases. The test cases cover a wide range of functionality of the classes under test.

The tools-package

The `tools`-package provides tools for logging events that occur during the run of the learners. The class `Logger` logs arbitrary messages and objects to a given output stream. The class `Statistics` writes statistics and information about the run of a learner to a specified file.

Workflow of the software

The software provides a command line interface to interact with the user. It is located in the `Learner`-class. When the software is executed, the `Learner`-class' main method is invoked. It parses the command line options and constructs a `DTDOracle`-type object for the requested DTD d. This object represents a MAT. As the main part of its construction, the automaton $\mathcal{A}_d$ as a `PDA`-type object is constructed by the `DTD`-class.

- To answer membership queries, the `membership`-method of the MAT is executed. It takes a `String` as parameter.
- To answer equivalence queries, the `isEquivalent`-method is invoked with an `FSA`-type object as parameter. This method uses the saturation algorithm located in the class `SaturationAlgorithm`.

After the construction of the MAT, the requested learning algorithm is invoked with the `DTDOracle`-type object as parameter. Each of the learning algorithms is realized as a `static` method:

- The $L^*_?$ and the $L^*_{?\ V2}$ learner are located in the package `angluin`, both realized in the class `LearningAlgorithmAngluin`.
- The RPNI$_?$ learner is realized in the class `LearningAlgorithmRPNI` located in the `rpni`-package.

Each method returns an `FSA`-type object as result. This object is the learned DFA, which is finally presented to the user.

4.3.2 Experiments & Analysis

Next, we present results and analyses of our software and try to deduce facts about the behavior of the heuristic algorithms in general.

Experimentation setup

All experiments were done on the RWTH Aachen high-performance computing cluster (HPC cluster).[4] The operating system used was Windows Server 2003 Compute Cluster Edition with Service Pack 2. Moreover, we used SUN's 64-bit Java SE Runtime Environment (build 1.6.0_02-ea-b02).

We chose to do our experiments on the HPC cluster since we expected the SAT solver of the $L^*_?$ learner and the $L^*_{?\ V2}$ learner to require a huge amount of main memory. This is also the reason why we used the 64-bit version of the Java runtime environment. It was, thus, possible to avoid the 4 GB main memory limit of 32-bit applications. However, parallelization is not supported by our implementation.

The Java virtual machine was executed with the options `-Xmx16G -Xms100M`. The first option `-Xmx16G` increases the maximum heap size available to the virtual machine to 16 GB. The second option `-Xms100M` sets the minimum heap size to 100 MB. All learners were executed without additional options except

[4]More information about the RWTH Aachen high-performance computing cluster can be found at `http://www.rz.rwth-aachen.de/hpc/`

$$d_1 = a \to a^? \qquad d_2 = \begin{array}{l} a \to o^? \\ o \to a \end{array} \qquad d_3 = \begin{array}{l} r \to a^* \\ a \to b \mid c \end{array}$$

$$d_4 = \begin{array}{l} r \to a^? \\ a \to b \\ b \to a^? \end{array} \qquad d_5 = \begin{array}{l} a \to o^* \\ o \to a^* \end{array} \qquad d_6 = \begin{array}{l} r \to abc \\ a \to ad \mid \varepsilon \\ b \to a \mid b \mid \varepsilon \\ c \to b \mid c \mid \varepsilon \\ d \to dc \mid \varepsilon \end{array}$$

$$d_7 = \begin{array}{l} r \to abc \\ a \to bfdc \\ b \to d \\ c \to ead \mid efd \\ d \to f \mid a \\ e \to b \\ f \to b \mid \varepsilon \end{array} \qquad d_8 = \begin{array}{l} r \to a^* \\ a \to b \end{array}$$

Figure 4.9: The DTDs used for experiments and analysis

the `-s` option to enable statistical logging. However, since all statistical data is immediately written to a file, the `-s` option does not have significant influence on the execution speed or any other runtime properties.

The disadvantage of the HPC cluster is that it is no single user environment and that we do not have any influence on the Job Manager. Moreover, we do not know how much processes run in parallel on a processor or how big the load is. Thus, we forgo to measure the runtime of our computations. This is also reasonable against the background of the frequent execution of the SAT solver by the $L^*_?$ and $L^*_{?\ V2}$ learners. Moreover, the runtime of the MAT also heavily influences the total runtime of a learner. Thus, only accumulated runtime measurements would be possible. However, the exact results presented later on are specific to the HPC cluster and may slightly differ on a single user environment.

In contrast to the learner relying on Angluin's algorithm, the RPNI$_?$ learner consumes only few memory. However, some of its runs took extremely long until they exceeded the time limit of ten days and had to be aborted.

The DTDs used in the experiments are shown in Figure 4.9. The DTDs d_1, d_2 and d_8 were mainly used for testing purpose during the programming of the software. However, they also produce good result. We use DTD d_8 to demonstrate typical runs of the learning algorithms. The DTDs d_3, d_4, d_5 and d_6 are taken from Segoufin's and Vianu's paper [SV02]. Thereby, DTD d_3 matches Example 3.2 of their paper, DTD d_4 matches Example 4.2, DTD d_5 matches Example 4.3 and DTD d_6 matches Example 4.11. The remaining DTD d_7 was provided by Segoufin and Sirangelo. Note that all DTDs are recognizable.

DTD	$L^*_?$	$L^*_{?\ V2}$	RPNI$_?$
d_1	✓		✓
d_2	✓		✓
d_3		✓	✓
d_4			✓
d_5	✓		✓
d_6			✗
d_7			✗
d_8	✓	✓	✓

Table 4.1: Results of the learning algorithms on the DTDs d_1 to d_8. Learned DTDs are marked with "✓" while blank entries indicate DTDs that could not be learned. A "✗" indicates an abortion due to exceeding the time limit

Results of the experiments

We ran all three heuristic algorithms, the $L^*_?$ learner, the $L^*_{?\ V2}$ learner and the RPNI$_?$ algorithm, on each of the eight DTDs. The results of these runs are shown in Table 4.1.

As one can see, both the $L^*_?$ and the $L^*_{?\ V2}$ learner were unable to learn an equivalent DFA for nearly half of the DTDs. This was because the SAT solver ran out of memory. Figure 4.10 shows the peak values of created variables and clauses for both learners on the DTDs where no equivalent DFA could be learned. The DTDs on the left side of the dashed line are those for which no equivalent DFA could be learned by the $L^*_?$ learner. The ones right to the dashed line are those for which the $L^*_{?\ V2}$ learner could not learn equivalent DFAs. Note that in the case of the $L^*_{?\ V2}$ learner, the peak values of the number of variables and the number of clauses do not need to occur simultaneously. However, the exact values of Figure 4.10 are only intended to get an idea of the sizes of the SAT instances. Obviously, the more memory is available the more variables and clauses can be generated and the bigger SAT instances can be solved. Thus, the exact peak values are only of minor interest.

The RPNI$_?$ algorithm achieved better results in our experiments than the algorithms based on Angluin's learner. However, it was also unable to learn equivalent DFAs for the more complex DTDs d_6 and d_7. The RPNI$_?$ learner did not run out of memory but exceeded the time limit and was aborted. Surprisingly, the equivalent DFAs for the DTDs d_1 to d_5 and the DTD d_8 could be learned with at most seven words in S_+ and seven words in S_-.

Analysis

In this analysis, we discuss typical runs of our software. We choose DTD d_8 for this purpose since it was learned by all algorithms and, furthermore, the runs of the learners show a typical behavior on this DTD.

The result of the $L^*_?$ learner on DTD d_8 is shown in Figure 4.11. As one can see, it is hard to tell whether this DFA really validates DTD d_8. This is caused by the fact that the SAT solver provides arbitrary models and no models that produce "intuitive" conjectures. However, it is easier to see that the DFA accepts all words in $L(d_8)$.

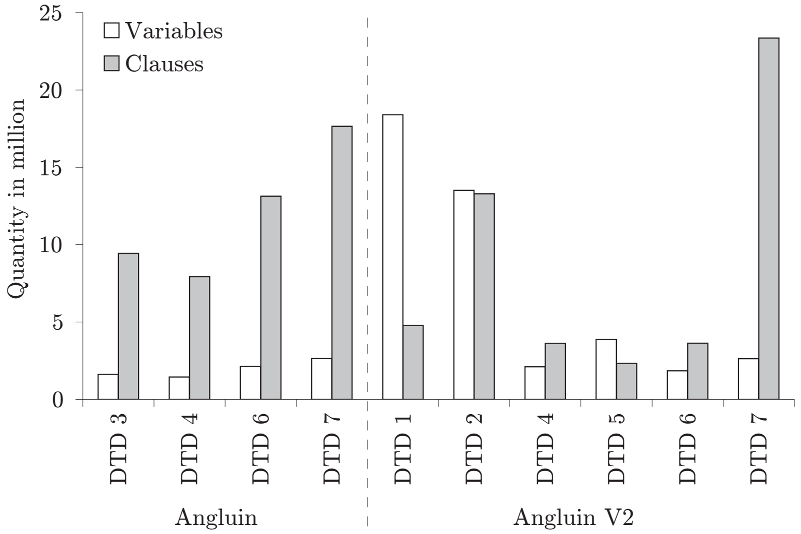


Figure 4.10: Peak values of the number of variables and clauses generated by the $L_?^*$ (Angluin) and $L_{?\ V2}^*$ (Angluin V2) learners on DTDs for which no equivalent DFA could be learned

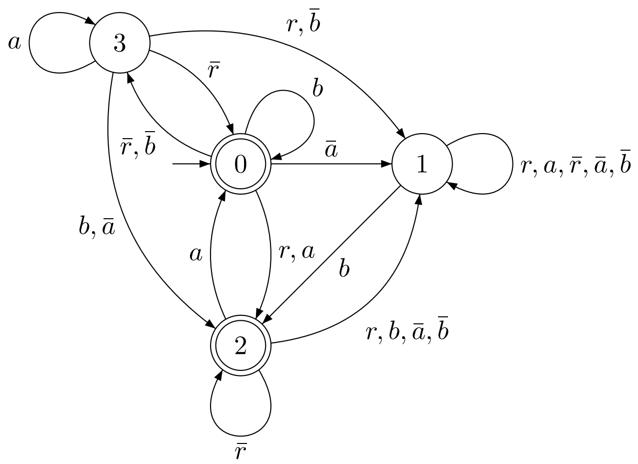


Figure 4.11: The result of the $L_?^*$ learner on DTD d_8. It is hard to verify that this DFA is equivalent to d_8. However, valid XML word representations are processed by the states 0, 2 and 3. The state 1 is a kind of sink state. Once reached this state, the DFA does not accept any XML word representation

The development of the total number of table entries, the number of ?-entries and their ratio during the run of the $L^*_?$ learner is depicted in Figure 4.12(a). The total number of table entries corresponds to the number of elements in the domain of the mapping T. The ratio shown in the figure is defined as

$$\frac{\text{Number of ?-entries}}{\text{Total number of table entries}} .$$

The most important observation we can deduce from Figure 4.12(a) is the fact that a huge part of the entries in the incomplete observation table are ?-entries. After the tenth loop, the ratio becomes nearly constant at approximately $0,98$. This can be explained by the following two observations:

- Since every tree node is represented by an opening and closing tag in its XML word representation, XML word representations are always of even length. Thus, every word of odd length in the table is no XML word representation and represented by a ?-entry.
- Besides the set R of representatives, the incomplete observation table also stores the set $R \cdot \Sigma$. This results in many representatives that cannot be completed to XML word representations since their last two symbols are no valid pair.

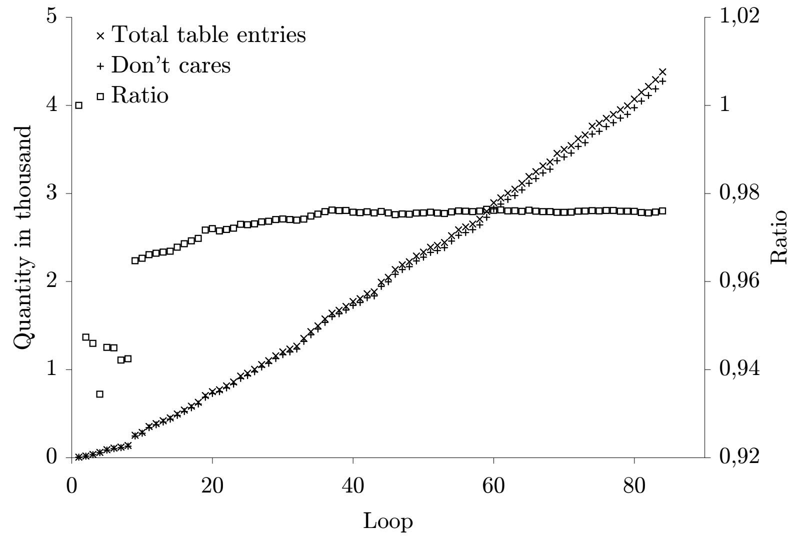
The huge ratio of ?-entries implies that only very little information about the DTD is actually stored in the table. The SAT solver nearly always computes models that make the table closed and consistent but do only slightly respect the DTD. In fact, in all except the eighth iteration of the run on DTD d_8, a model for the formula $\psi_{cl} \wedge \psi_{co}$ could be computed. The resulting conjecture was not equivalent and caused a new counter-example to be added. This increased the size of the table and, hence, the number of variables and clauses generated in the next iteration. This behavior can be seen in Figure 4.12(b). Note that only the number of variables and clauses of the formula $\psi_{cl} \wedge \psi_{co}$ are shown. The eighth loop was the only one in which neither a model for $\psi_{cl} \wedge \psi_{co}$ nor for ψ_{co} could be found. However, a model for ψ_{cl} was computed, which caused a new sample to be added. One can see the results of this insertion as leap in the size of the table in the lower left part of Figure 4.12(a).

We deduce an interesting general observation from this behavior.

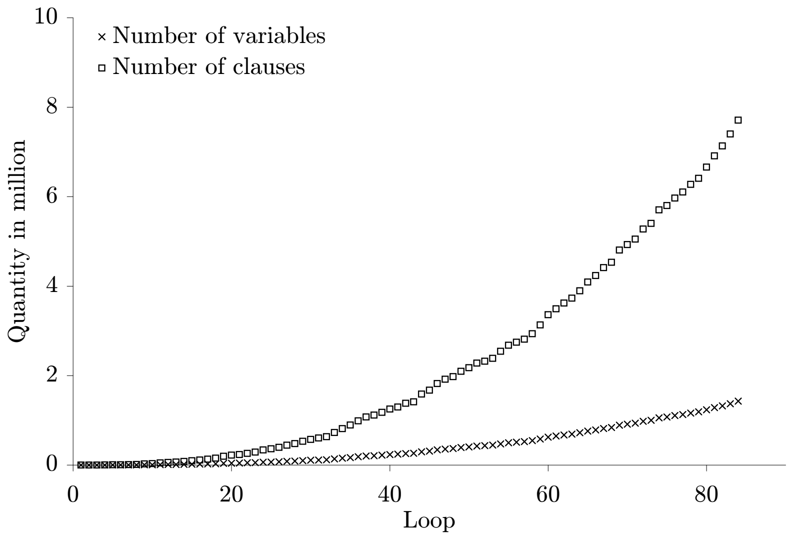
Observation 4.23. Let us assume that the incomplete observation table has a certain number of samples, say n. As our experiments show, the $L^*_?$ learner successively computes models $\mu \models \psi_{cl} \wedge \psi_{co}$, asks a membership query on the resulting conjecture and adds a counter-example (if necessary) until no further model can be found. Then, it adds a new sample (n is thereby increased by one) and starts over again.

◁

Let us consider this observation from a slightly different perspective. If the table has n samples, then there can be at most 2^n states of a conjecture since there are at most 2^n different rows with n Boolean entries. A model $\mu \models \psi_{cl} \wedge \psi_{co}$ results in a conjecture that is compatible with the information stored in the table. This conjecture is possibly not equivalent and, thus, the MAT returns a counter-example on an equivalence query. If this counter-example is added to



(a) The development of the number of table entries, the number of ?-entries and their ratio during the run of the $L^*_?$ algorithm on DTD d_8. The size of the table and the number of ?-entries are shown on the left y-axis while the ratio of both is shown on the right y-axis



(b) The development of the number of variables and clauses generated by the formula $\psi_{cl} \wedge \psi_{co}$ during the run of the $L^*_?$ learner on DTD d_8

Figure 4.12: A detailed view of the run of the $L^*_?$ learner on DTD d_8

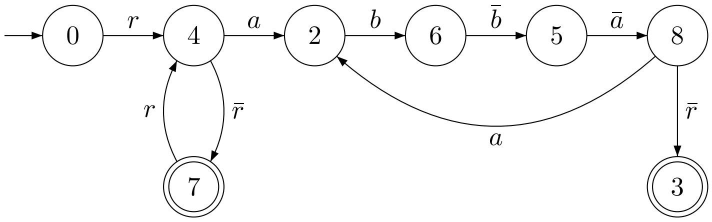


Figure 4.13: The result of the $L^*_{?\ V2}$ learner on DTD d_8. The sink state 1 and its incoming transitions are not shown. It is easy to see how this DFA performs the validation. The DFA forces every accepted word to start with r and end with $\bar{r}$. In between, it allows an arbitrary number of repetitions of the word $ab\bar{b}\bar{a}$

the table, then the next conjecture treats this counter-example correctly. This repeats until there is no new model. One can think of this as an improved trail and error procedure: The learner tries every possible conjecture with 2^n or less states that is compatible with the stored data. This repeats until no compatible conjecture with at most 2^n states is found. Then, the learner adds a new sample, which increases n and, thus, the number of possible states of a conjecture. Therefore, the learner successively checks all compatible DFAs in increasing size. Clearly, this is no efficient procedure. It is, therefore, not astonishing that the learner did not perform well in the experiments.

We do not know whether this process always terminates but Observation 4.23 may be used to show the termination of the $L^*_?$ learner. Thereto, it is necessary to prove that helpful samples are added, which allow separating the equivalence classes of a suitable target language. Unfortunately, due to the high memory consumption, we were not able to verify the termination empirically.

Grinchtein and Leucker [GL] propose a similar approach in a not yet published paper. Their algorithm is a combination of Angluin's algorithm and a method proposed by Biermann and Feldman [BF72]. Grinchtein and Leucker's algorithm uses a SAT solver to compute a minimal DFA compatible with a learned part of the target language. Starting with a DFA with one state, they check successively all DFAs with at most n states and increment the value of n if no equivalent DFA is found. They were able to show that this procedure guarantees termination. Since their algorithm resembles our $L^*_?$ learner, we are hopeful that it is also possible to prove the termination of the $L^*_?$ learner.

The result of the $L^*_{?\ V2}$ learner on DTD d_8 is shown in Figure 4.13. This time the resulting DFA is much more intuitive. However, this is no general observation. In fact, the DFAs learned by the $L^*_{?\ V2}$ learner are in general much bigger than the ones learned by the original version. This is caused by the fact that the ?-entries are replaced with the values of a model.

Since we consider the SAT solver to be a black box and, hence, have no influence on the computed model, it is much harder to make general statements about this learner. However, we state that it is more unlikely to find models for the formula $\psi_{cl} \wedge \psi_{co}$ since the percentage of ?-entries to the size of the table is much smaller than for the $L^*_?$ learner. Thus, the formulae ψ_{cl} and ψ_{co} are also often constructed.

DTD	$L^*_?$	$L^*_{?\ V2}$	RPNI$_?$
d_1		✓	
d_2		✗	
d_3	✓		
d_4	✓	✗	
d_5		✗	
d_6	✗	✗	✗
d_7	✗	✗	✗
d_8			

Table 4.2: Results of the learning algorithms with enabled valid pair improvement. A "✓" stands for a successfully learned DTD while a "✗" stands for a failure either because of to few main memory or an exceeded runtime limit. A blank entry in the table means that this experiment was not done

Figure 4.14 shows details of the run of the $L^*_{?\ V2}$ learner on DTD d_8. The total number of table entries, the number of ?-entries and their ratio are shown in Figure 4.14(a). As expected, the ratio is in average much lower than in the case of the original learner. The development of the number of variables and clauses of the formulae are shown in Figure 4.14(b). The large leap of all values in the ninth loop was caused by the insertion of a higher-than-average long counter-example.

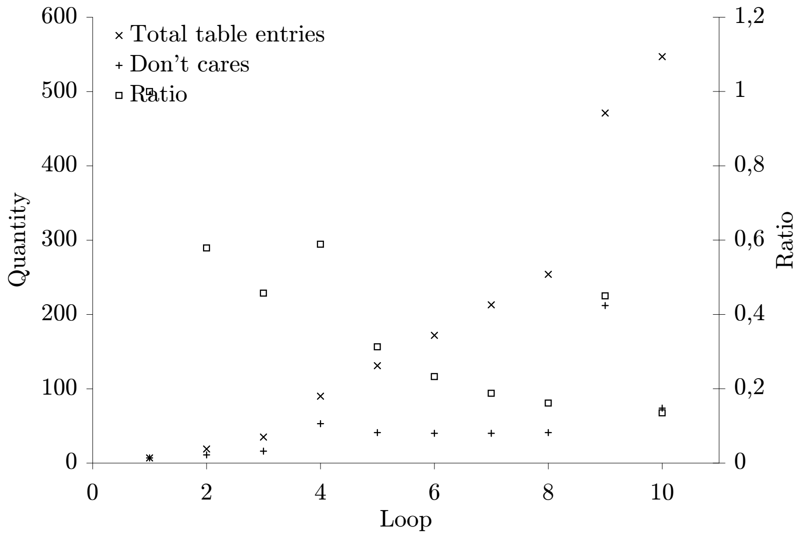
Figure 4.15 shows the result of the RPNI$_?$ algorithm on DTD d_8. Again, this DFA is more "intuitive" than the result of the $L^*_?$ learner. However, also for the RPNI$_?$ algorithm it is hard to deduce general statements. This is mainly because the RPNI$_?$ algorithm does only rely on equivalence queries on which the MAT's answer is not unique.

We did experiments with the valid pair improvement only for those DTDs for which a basic algorithm failed learning an equivalent DFA. The results on these DTDs are shown in Table 4.2. Again, the time limit was ten days.

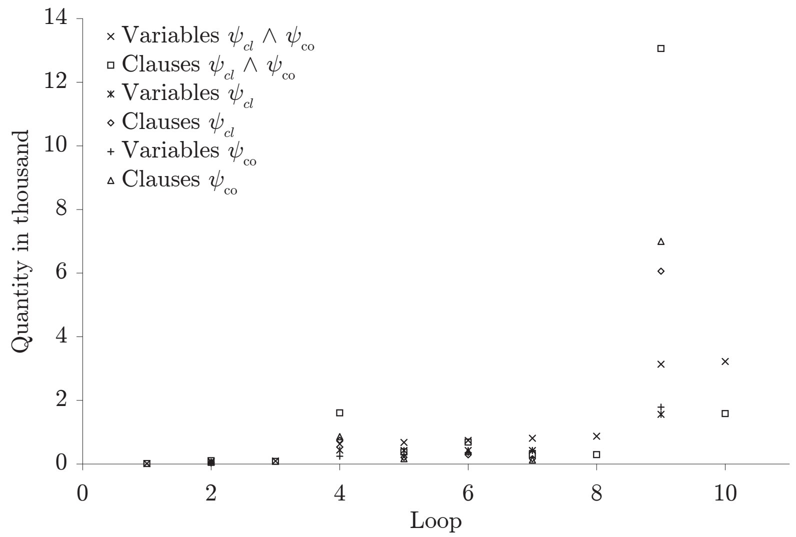
We observe that, in the case of the $L^*_?$ learner, the valid pair improvement allowed to learn two more DTDs. Moreover, the valid pair improvement allowed the $L^*_{?\ V2}$ algorithm to learn one additional DTD. The more complex DTDs d_6 and d_7 could not be learned in any case.

However, let us qualify these more positive results:

- Due to the valid pair improvement, the runtime of the MAT increased significantly: Recall that the runtime of an equivalence query is $O(|Q_\mathcal{A}|^5)$ where $|Q_\mathcal{A}|$ is the size of the state set of the provided conjecture. Since the valid pair improvement constructs the conjecture $\mathcal{A}_{VP_d} \otimes \mathcal{A}$, the size of the conjecture and, hence, the runtime of equivalence queries is increased by the factor $|Q_{\mathcal{A}_{VP_d}}|^5$. Thereby, the size of the DFA $\mathcal{A}_{VP_d}$ is determined by the number of tags of the DTD because $\mathcal{A}_{VP_d}$ uses the opening and closing tags as its states, i.e. $|Q_{\mathcal{A}_{VP_d}}| = 2 \cdot \Sigma_{tag} + 1$. Note that the factor is constant but can become large even for small DTDs.



(a) The development of the total number of table entries, the number of ?-entries and their ratio during the run of the $L^*_{?\ V2}$ learner on DTD d_8. The total number of table entries and the number of ?-entries are shown on the left y-axis while the ratio of both is shown on the right y-axis



(b) The development of the number of variables and clauses generated during the run of the $L^*_{?\ V2}$ learner on DTD d_8

Figure 4.14: A detailed view of the run of the $L^*_{?\ V2}$ algorithm on DTD d_8

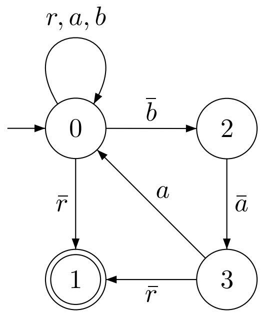


Figure 4.15: The result of the RPNI$_?$ algorithm on DTD d_8. The sink state 4 and its incoming transitions are not shown. It is not hard to see how this DFA performs the validation. The state 0 is used to read the opening tags while the states 2 and 3 are used to read the closing ones

- In our experiments the DFAs $\mathcal{A}_{VP_d}$ already contains sufficient information about the DTD to construct equivalent conjectures. The valid pair DFAs for the DTDs d_1, d_2, d_3, d_4, d_5 and d_8 are already equivalent to their DTD respectively. Thus, it is not astonishing that the learners are able to learn additional DTDs. However, note that the valid pair improvement does not allow to learn the more complex DTDs d_6 and d_7.

Conclusion

Our implementation is intended to be a proof-of-concept and shall show that the algorithms developed in this chapter are able to learn DFAs for validating XML word representations against DTDs. As shown, our learners succeeded in learning DFAs for the simple DTDs but DFAs for the more complex DTDs could not be learned. This result also holds when using high-performance computing. Because of this negative result, we refrained from using real world application DTDs in our experimentation.

Due to the high memory consumption of the $L^*_?$ and $L^*_{?\ V2}$ learner and the high runtime of the RPNI$_?$ algorithm it is doubtful whether our learners can be used in practice. However, there are some possible improvements to the implementation. These may yield results that are more positive.

- The use of C or C++ as programming language may result in a notable enhancement of speed. Moreover, our implementation uses some of Java's standard libraries and data structures. More suited data structures may also increase the speed.

- An additional enhancement may be enjoyed by using a parallelized SAT-solver. Thus, it would be possible to use the whole capabilities of the HPC cluster.

- A SAT-solver that uses the hard drive instead of the main memory for its computations removes the restriction of limited main memory. However, it is unlikely that this results in an acceptable runtime.

- Finally, other heuristics in the case that there is neither a model for ψ_{cl} nor for ψ_{co} may yield better results.

5 Validating parenthesis word representations

In this chapter, we consider the validation of trees given as parenthesis words representations (cf. Definition 2.12) against a DTD. As in Chapter 4, our final objective is the learning of DFAs that are capable of validating parenthesis word representations. Thereby, we assume that, analogous to the case of validation XML word representations, those DFAs are only provided with valid parenthesis word representations. Again, we do not care whether the learned DFA accepts or rejects non parenthesis word representations. Formally we say that a DTD $d = (\Sigma_{tag}, r, \rho)$ *can be validated* by a DFA $\mathcal{A}$ if and only if

$$L^{()}(d) = L(\mathcal{A}) \cap L^{()}$$

holds. In this case, we call d *recognizable* and $\mathcal{A}$ *equivalent* to d.

Example 5.1. Reconsider the DTD $d = \begin{array}{l} r \to a \\ a \to a^? \end{array}$ from Example 4.1. This DTD is recognizable by a finite automaton over $\Sigma = \{r, a, (,)\}$, whose accepted language is defined by the regular expression $r([a(]^+)^+$.

$\triangleleft$

We argue in a moment that validation of parenthesis word representations using DFAs on the one hand and 1-VCAs, working over the pushdown alphabet $\Sigma_c = \{(\}$, $\Sigma_r = \{)\}$ and $\Sigma_{int} = \Sigma_{tag}$, on the other is closely related. Such 1-VCAs use their counter to count the occurrences of opening and closing parenthesis and can decide whether the provided input is a parenthesis word representations. Hence, we require the 1-VCA to also check the input to be a parenthesis word representation. We say that an 1-VCA $\mathcal{A}$ is *equivalent to* d if

$$L(\mathcal{A}) = L^{()}(d) \ .$$

The connection between validating parenthesis word representations using DFAs and 1-VCAs can easily be seen. Intuitively an 1-VCA uses its counter only to check whether the input is a parenthesis word representation. Since, during the validation, the counter value of an 1-VCA is greater than 0 after the first two input symbols of a parenthesis word representation, the actual validation is done in a “regular” way. Theorem 5.2 states this observation formally.

Theorem 5.2. *A DTD $d = (\Sigma_{tag}, r, \rho)$ can be validated by an 1-VCA $\mathcal{A}$ working over the pushdown alphabet $\Sigma_c = \{(\}$, $\Sigma_r = \{)\}$ and $\Sigma_{int} = \Sigma_{tag}$ if and only if there is a DFA $\mathcal{B}$ working over the alphabet $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$ that validates d. Moreover, the DFA $\mathcal{B}$ can be constructed efficiently from the 1-VCA $\mathcal{A}$ and vice versa.*

Before we turn on proving Theorem 5.2, let us remark the use of this theorem. The overall objective is to provide a learning algorithm for DFAs that are capable of validating parenthesis word representations against DTDs. Since the recognizability of a DTD d can be expressed equivalently via DFAs or 1-VCAs, we can now choose which automata model we want to use in a learning algorithm. Moreover, after the learning of either a DFA or an 1-VCA, we can efficiently transform these automata into the other automata model respectively.

However, the learning of 1-VCAs has an essential advantage: We do no longer need to deal with "don't cares" since an 1-VCA is capable of deciding whether the input is a parenthesis word representation. This overcomes the problems that arise in Chapter 4. In this chapter we, therefore, concentrate on learning 1-VCAs for the validation of parenthesis word representations.

It turns out that some intractable difficulties complicate the direct learning of 1-VCAs. Hence, we develop a more general algorithm to learn m-VCA acceptable languages. A method proposed by Bárány, Löding and Serre [BLS06] to reduce the threshold m can then be applied to the resulting m-VCA yielding an 1-VCA.

This chapter is organized as follows. In Section 5.1 we briefly discuss a MAT for validating parenthesis word representations against a DTD and have a closer look at the question whether it is decidable if a DTD is recognizable in this context. In Section 5.2 we present the difficulties that complicate the direct learning of 1-VCAs. Also in this section, we introduce a method to learn real time one-counter automata, which is the basis for our learning algorithm. Finally, in Section 5.3 we develop our algorithm to learn VCA acceptable languages and describe its application to the setting of validating parenthesis word representations.

Let us now turn on proving Theorem 5.2. For this proof it is useful to have a closer look on how a parenthesis word representation is validated by an 1-VCA. Thus, let $d = (\Sigma_{tag}, r, \rho)$ be a DTD and consider an 1-VCA $\mathcal{A}$ working over the pushdown alphabet $\Sigma_c = \{(\}$, $\Sigma_r = \{)\}$ and $\Sigma_{int} = \Sigma_{tag}$ with $L(\mathcal{A}) = L^{()}(d)$. Let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$.

We first observe that every valid parenthesis word representations is of the form

$$w = r(w')$$

where w' is a (possible empty) sequence of parenthesis word representations of subtrees of the root node r. Moreover, we observe that the prefixes ε and r have a counter value of 0 while all other proper prefixes have a counter value greater than 0. Thus, an accepting run of an 1-VCA $\mathcal{A}$ is always of the form

$$\underbrace{(q_0^{\mathcal{A}}, 0) \xrightarrow{r}_{\mathcal{A}} (q, 0) \xrightarrow{(}_{\mathcal{A}}}_{\delta_0^{\mathcal{A}}} \underbrace{(q', 1) \xrightarrow{w')}_{\mathcal{A}} (q'', 0)}_{\delta_1^{\mathcal{A}}}$$

where $q'' \in F_{\mathcal{A}}$. On this run $\delta_0^{\mathcal{A}}$-transitions are only used in the first two steps while $\delta_1^{\mathcal{A}}$-transitions are exclusively used in the rest. Moreover, we observe that the input is either no parenthesis word representation or not valid w.r.t. d if the run of $\mathcal{A}$ on the input is not of this special form.

We exploit this observation to construct a DFA $\mathcal{B}$ capable of validating d from the 1-VCA $\mathcal{A}$. Our idea is to use $\mathcal{A}$'s states to simulate the last part of the run where only $\delta_1^{\mathcal{A}}$-transitions are applied. To simulate the first two steps, we add two new states $q_0^{\mathcal{B}}$ and $q_1^{\mathcal{B}}$ and force the input to start with $r($.

Construction 5.3. *Let d be a DTD and $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta_0^{\mathcal{A}}, \delta_1^{\mathcal{A}}, F_{\mathcal{A}})$ be an 1-VCA with $L(\mathcal{A}) = L^{()}(d)$. The DFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta^{\mathcal{B}}, F_{\mathcal{B}})$ is defined as*

- $Q_{\mathcal{B}} = Q_{\mathcal{A}} \cup \{q_0^{\mathcal{B}}, q_1^{\mathcal{B}}\}$,
- $F_{\mathcal{B}} = F_{\mathcal{A}}$ *and*
- $\delta^{\mathcal{B}}$ *defined by*

$$\begin{aligned}
&\delta^{\mathcal{B}}(q_0^{\mathcal{B}}, r) = q_1^{\mathcal{B}} \ , \\
&\delta^{\mathcal{B}}(q_1^{\mathcal{B}}, () = \delta_0^{\mathcal{A}}(\delta_0^{\mathcal{A}}(q_0^{\mathcal{A}}, r), () \ , \\
&\delta^{\mathcal{B}}(q, a) = \delta_1^{\mathcal{A}}(q, a) \textit{ for all } q \in Q_{\mathcal{A}} \textit{ and } a \in \Sigma \ .
\end{aligned}$$

Note that this construction yields a partial DFA, which can be completed by adding a sink state.

We claim that $\mathcal{B}$ is also capable of validating d if the input is assumed to be a parenthesis word representation.

Lemma 5.4. *Let $\mathcal{A}$ be an 1-VCA equivalent to a DTD d and $\mathcal{B}$ be the DFA constructed above. Then,*

$$L(\mathcal{A}) = L(\mathcal{B}) \cap L^{()}$$

holds.

Proof of Lemma 5.4. Let $w \in L(\mathcal{A})$. Then, w is of the form $w = r(w')$. Moreover, there is a run

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{r}_{\mathcal{A}} (q, 0) \xrightarrow{(}_{\mathcal{A}} (q', 1) \xrightarrow{w')}_{\mathcal{A}} (q'', 0) \ .$$

From the observation about accepting runs of $\mathcal{A}$ we know that $\delta_0^{\mathcal{A}}$-transitions are applied in the first two steps of this run. Hence,

$$q_0^{\mathcal{B}} \xrightarrow{r}_{\mathcal{B}} q_1^{\mathcal{B}} \xrightarrow{(}_{\mathcal{B}} q'$$

also holds. Moreover, we deduce that there exists a run

$$q' \xrightarrow{w')}_{\mathcal{B}} q''$$

since this is an exact copy of the run of $\mathcal{A}$.[1] Since $F_{\mathcal{A}} = F_{\mathcal{B}}$, this run is accepting. Thus, $w \in L(\mathcal{B})$. Moreover, we know that $L(\mathcal{A}) \subseteq L^{()}$ holds and, thus, $L(\mathcal{A}) \subseteq L(\mathcal{B}) \cap L^{()}$.

Now, assume that $w \in L(\mathcal{B}) \cap L^{()}$. Since $w \in L^{()}$, it is of the form $w = r(w')$ where w' is a (possible empty) sequence of parenthesis word representations of subtrees. Moreover, there exists a run

$$q_0^{\mathcal{B}} \xrightarrow{r}_{\mathcal{B}} q_1^{\mathcal{B}} \xrightarrow{(}_{\mathcal{B}} q' \xrightarrow{w')}_{\mathcal{B}} q''$$

[1]Formally, we can show this by induction over the length of the run. However, since the states and transitions of $\mathcal{A}$ are exactly copied, we skip this proof here

with $q'' \in F_{\mathcal{B}}$. From the observation about runs of 1-VCAs on parenthesis words representations we know that $cv(u) \geq 0$ for all prefixes u of w. Additionally, $cv(\varepsilon) = cv(r) = 0$ and $cv(u) > 0$ for all other proper prefixes of w. Thus, by definition of $\mathcal{B}$ and with the same argumentation as before we conclude that there exists a run

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{r}_{\mathcal{A}} (q, 0) \xrightarrow{(}_{\mathcal{A}} (q', 1) \xrightarrow{w')}_{\mathcal{A}} (q'', 0)$$

for a $q \in Q_{\mathcal{A}}$. Since $F_{\mathcal{A}} = F_{\mathcal{B}}$, this run is accepting and $w \in L(\mathcal{A})$.

All in all, we gain $L(\mathcal{A}) = L(\mathcal{B}) \cap L_{()}$.

□

Note that the transformation of an 1-VCA into a DFA can be done in polynomial time. In fact, we just need to copy the states and transitions of $\mathcal{A}$ and add two new states. The resulting DFA has $|Q_{\mathcal{A}}| + 2$ states and we can perform the transformation in $O(|Q_{\mathcal{A}}| \cdot |\Sigma|)$.

Let us now consider the converse direction, i.e. the construction of an 1-VCA from a DFA equivalent to d. Hence, consider a DFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_0^{\mathcal{B}}, \delta^{\mathcal{B}}, F_{\mathcal{B}})$ with $L(\mathcal{B}) \cap L_{()} = L(d)$. The idea of this construction is the same as for the previous construction. We use two new states to simulate the first two transitions by using $\delta_0^{\mathcal{A}}$. The simulation of the last part of the run is again done by a copy of the states and transitions of $\mathcal{B}$. In this copy we only allow $\delta_1^{\mathcal{A}}$-transition.

Construction 5.5. *We define the* 1*-VCA* $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta_0^{\mathcal{A}}, \delta_1^{\mathcal{A}}, F_{\mathcal{A}})$ *as follows.*

- $Q_{\mathcal{A}} = Q_{\mathcal{B}} \cup \{q_0^{\mathcal{A}}, q_1^{\mathcal{A}}\}$,
- $F_{\mathcal{A}} = F_{\mathcal{B}}$ *and*
- $\delta_0^{\mathcal{B}}$ *and* $\delta_1^{\mathcal{B}}$ *defined by*

$$\begin{aligned}
&\delta_0^{\mathcal{A}}(q_0^{\mathcal{A}}, r) = q_1^{\mathcal{A}} \ , \\
&\delta_0^{\mathcal{A}}(q_1^{\mathcal{A}}, () = \delta^{\mathcal{B}}(\delta^{\mathcal{B}}(q_0^{\mathcal{B}}, r), () \ , \\
&\delta_1^{\mathcal{A}}(q, a) = \delta^{\mathcal{B}}(q, a) \textit{ for all } q \in Q_{\mathcal{B}} \textit{ and } a \in \Sigma \ .
\end{aligned}$$

Again, note that $\mathcal{A}$ is a partial 1-VCA, which can be completed by adding a sink state.

The next lemma shows that the construction is correct. However, the proof is analogous to the proof of Lemma 5.4 and, hence, skipped.

Lemma 5.6. *Let* $\mathcal{B}$ *be a DFA equivalent to a DTD* d *and* $\mathcal{A}$ *be the* 1*-VCA constructed above. Then,*

$$L(\mathcal{B}) \cap L_{()} = L(\mathcal{A})$$

holds.

As a last remark, we state that the resulting 1-VCA has $|Q_{\mathcal{B}}| + 2$ states and the transformation can be done in $O(|Q_{\mathcal{B}}| \cdot |\Sigma|)$. Moreover, we are now able to prove Theorem 5.2.

Proof of Theorem 5.2. Theorem 5.2 follows directly from Lemma 5.4 and 5.6.

□

5.1 The decision problem for parenthesis word representations

The decision problem for parenthesis word representations is the following problem:

"Given a DTD d. Is d recognizable?"

In contrast to the decision problem for XML word representations, this decision problem is decidable. To show this, we first construct an eVPA $\mathcal{A}_d^{()}$, which accepts exactly all parenthesis word representations that are valid w.r.t. a given DTD $d = (\Sigma_{tag}, r, \rho)$. The idea of this construction is analogous to the construction of the eVPA $\mathcal{A}_d$ for XML word representations in Section 4.1. The used pushdown alphabet is $\Sigma_{int} = \Sigma_{tag}$, $\Sigma_c = \{(\}$ and $\Sigma_r = \{)\}$ and, thus, $\Sigma = \Sigma_{int} \cup \Sigma_c \cup \Sigma_r$. The eVPA $\mathcal{A}_d^{()}$ uses its stack to simulate a horizontal DFA every time an internal symbol is processed. If it reads a call symbol, it pushes the initial state of the new horizontal DFA on the stack and, finally, pops the top stack symbol on reading a return symbol. Moreover, the eVPA $\mathcal{A}_d^{()}$ can decide whether the provided input is a parenthesis word representation. Note that the eVPA $\mathcal{A}_d^{()}$ can be used to construct a MAT for parenthesis word representations in the same way as the MAT for XML word representations.

The decidability of the above stated problem is a direct corollary of results of Bárány, Löding and Serre [BLS06]. In their paper, they were able to show that the two decision problems

(1) "Given a VPA $\mathcal{A}$, is there $m \in \mathbb{N}$ and an m-VCA that accepts $L(\mathcal{A})$?"

(2) "Given an m-VCA $\mathcal{A}$ and $m' \in \mathbb{N}$, is there an m'-VCA that accepts $L(\mathcal{A})$?"

are decidable. Moreover, they provided a method to construct both an m-VCA for question (1) and an m'-VCA for question (2). Unfortunately, these constructions are more than exponential in the number of states of the given automaton. To be precise, the upper bound for (1) is five times exponential while the upper bound for (2) is doubly exponential.

Regardless of these high complexities, it is now easy to gain the answer for the decidability problem for parenthesis word representations: First, we construct the eVPA $\mathcal{A}_d^{()}$ and transform it into a VPA by using Construction 2.18. Then, we use (1) to decide whether there exists an m-VCA $\mathcal{A}$ with $L(\mathcal{A}) = L(\mathcal{A}_d^{()}) = L^{()}(d)$. If this is the case, we construct this m-VCA. Finally, we use (2) to decide whether there exists an 1-VCA that accepts $L(\mathcal{A})$ where we set $m' = 1$. The answer to this question is the answer to the decision problem.

5.2 A closer look at learning 1-VCAs

In this section, we have a closer look at the direct learning of 1-VCA-acceptable languages and the problems that arise. We start by introducing an algorithm of Fahmy and Roos [FR95], by which our learning algorithm is inspired. In their paper they show how to learn *real time one-counter automata (ROCA)*.[2]

[2] A real time one-counter automaton is an one-counter automaton without ε-transitions that accepts with final states and a counter value of 0. Moreover, it can check whether its counter has value 0 or not

Thereby, they use an improvement of an algorithm proposed by Fahmy and Biermann [FB93]. Fahmy and Roos' algorithm uses two tiers:

1. First, a sufficient large initial fragment B_n of the behavior graph BG_L of the target language L is learned. This initial fragment consists of all states that have distance n or less from the initial state of BG_L. The actual learning is done by an "appropriately modified" version of Angluin's learning algorithm for regular languages. However, it remains open how this appropriate modification looks like and how it can be guaranteed that BG_n is sufficiently large.

2. Then, the initial fragment B_n is decomposed into a control structure (the automaton) and a data structure (its counter).

Note that the complete learning is performed in the first step. After constructing the initial fragment of the behavior graph, no more queries to the MAT are necessary.

The second tier needs to be discussed in more detail. Fahmy and Roos state the following fundamental fact about ROCA-acceptable languages.

Observation 5.7. For every ROCA-acceptable language L there is a ROCA $\mathcal{A}$ such that the behavior graph BG_L is isomorphic to the configuration graph $G_{\mathcal{A}}$. ◁

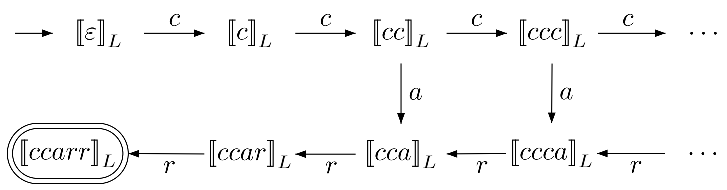
In other words Observation 5.7 states that one can think of each state of the behavior graph as a configuration of the ROCA $\mathcal{A}$, i.e. a state of $\mathcal{A}$ and an appropriate counter value. Thus, a ROCA recognizing L can immediately be obtained by properly identifying and decomposing the states and the counter values represented by the states of the behavior graph. This connection is shown in Figure 5.1. It is easy to see that the behavior graph of L in Figure 5.1(a) is isomorphic to the configuration graph of the ROCA depicted in Figure 5.1(b). The configuration graph itself is depicted in Figure 5.1(c).

To identify a suitable ROCA, Fahmy and Roos exploit a second property of ROCA-acceptable languages, which is stated in the following observation.

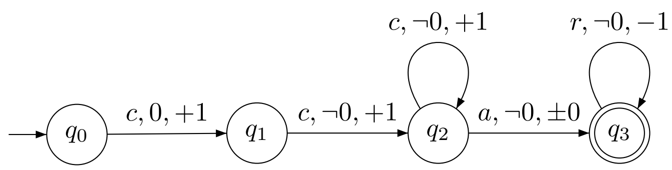
Observation 5.8. The behavior graph of a ROCA-acceptable language consists of an initial part and an isomorphic repeating structure. ◁

One can easily check this observation in the example in Figure 5.1(a). Intuitively, the fact that a ROCA can only keep track of its counter-value up to a certain threshold by using its states causes this repeating structure. If the threshold is exceeded, the ROCA works in a "regular way" and, thus, generates this repeating structure.

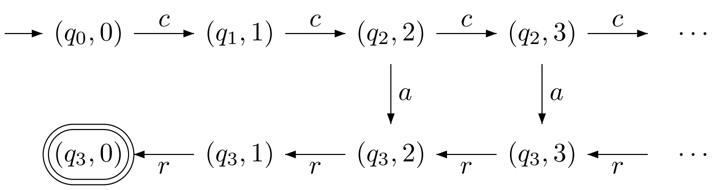
The repeating structure allows the (possibly) infinite behavior graph of a ROCA-acceptable language L to be represented by using only finite information. The idea of Fahmy and Roos is to construct a ROCA that simulates the run of the behavior graph on the given input. Thereto, they use the initial part and the first repetition of the behavior graph as states of a ROCA $\mathcal{A}$ such that $\mathcal{A}$ works like the behavior graph on these states. If $\mathcal{A}$ has to process an input that leads into the second repetition, $\mathcal{A}$ simulates this by changing its control state to the appropriate control state in the first repetition and incrementing the counter by one. Since the repeating structure is isomorphic, this can be repeated if the



(a) The behavior graph of the language L. The equivalence class that contains all words that cannot be completed to an accepting word is not shown



(b) A ROCA $\mathcal{A}$ accepting the language L. A transition $q \xrightarrow{a,x,y} q'$ with $a \in \Sigma$, $x \in \{0, \neg 0\}$ and $y \in \{+1, -1, \pm 0\}$ indicates that the ROCA can change its control state from q to q' on reading a while the counter value is 0 or not 0 respectively. Furthermore, the counter value is changed according to y. Note that $\mathcal{A}$ is a partial ROCA, which can easily be completed by adding a sink state



(c) The configuration graph of the ROCA $\mathcal{A}$. All configurations of the sink state of $\mathcal{A}$ are not shown

Figure 5.1: The repeating structure of the language $L = \{ccc^n ar^n rr \mid n \geq 0\}$ over the alphabet $\Sigma = \{c, a, r\}$. Moreover, the figure shows the connection between the behavior graph of L and the configuration graph of $\mathcal{A}$

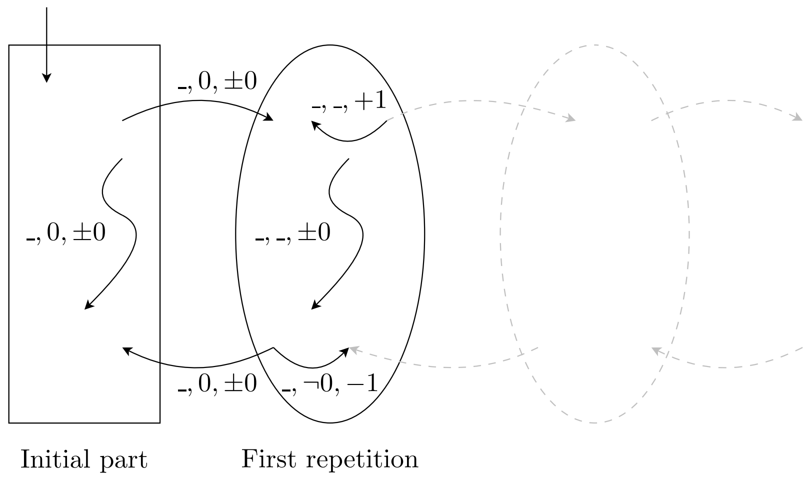


Figure 5.2: Fahmy and Roos' construction of a ROCA by exploiting the repeating structure of the behavior graph

input leads to a higher repetition. Thus, the counter is only used to keep track of the number of repetitions the ROCA is currently simulating.

Figure 5.2 shows a sketch of this construction. Within the initial part and the first repetition, the counter value is not changed. A change is only done if $\mathcal{A}$ simulates a step into the next higher or lower repetition. In the first case, $\mathcal{A}$ increases the counter value while it decreases the counter in the latter case. If $\mathcal{A}$ is currently in the repetitive part, a counter value of i indicates that the simulated run of the behavior graph is currently in the $(i+1)$-th repetition. If $i = 0$ and $\mathcal{A}$ leaves the repetitive part for the next lower one, it jumps into the initial part. Note that the ability of a ROCA to change its counter at will is essential for its function. For the detailed definition of the ROCA $\mathcal{A}$, we refer to Fahmy and Roos [FR95]. However, it is not hard to see that the behavior graph of L and the configuration graph of $\mathcal{A}$ are isomorphic.

It remains to show how the repeating structure can be identified. An efficient method is especially necessary since identifying isomorphic sub graphs is known to be NP-complete. However, in this particular case, Fahmy and Roos exploit the special structure of the behavior graph: The isomorphic repetitions occur successively. Their idea is to perform several *parallel breadth-first searches (PBFS)* to identify the isomorphic parts. This is sketched in Figure 5.3. If each of the parallel running breadth-first searches discovers the same structure, we know that those parts are isomorphic. However, to find the isomorphic subgraphs, it can be necessary to try several PBFS with different entry points. To determine those entry points, so called *exit points* are of special interest. Such exit points are equivalence classes that have a transition pointing out of the initial fragment of BG_L. The entry points are gained by decomposing a word that leads to an exit point into repeating infixes. Again, we refer to Fahmy and Roos [FR95] for a detailed description of the PBFS.

Fahmy and Roos showed the following runtime complexity of their learning algorithm.

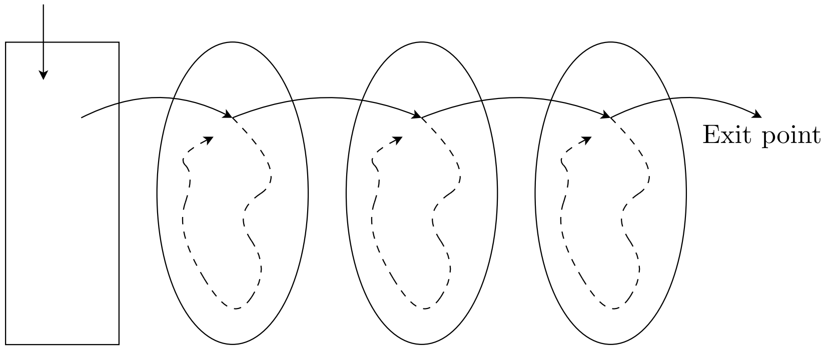


Figure 5.3: Performing a parallel breadth-first search to identify isomorphic sub graphs of the behavior graph of a ROCA-acceptable language

Theorem 5.9 (Fahmy and Roos [FR95])**.** *Let L be a ROCA-acceptable language. Both the learning of the initial fragment BG_n of the behavior graph BG_L and the construction of the ROCA can be done in time polynomial in the size, i.e. in the number of states, of the initial fragment of the behavior graph.*

However, the size of such an initial fragment can be exponential in the number of states of a minimal ROCA accepting the language L.

Our learning algorithm uses both the idea of using an initial fragment of the behavior graph and the identification of repeating structures in a modified way. Unfortunately, it turns out that the learning of 1-VCAs is involving and Fahmy and Roos' algorithm cannot be applied. Mainly, this is due to the following two facts.

- The behavior graph of an 1-VCA acceptable language does not contain enough information for a direct construction of an 1-VCA. Lemma 5.10 states this formally. It shows that there are 1-VCA-acceptable languages such that every 1-VCA accepting it has a configuration graph that is not isomorphic to the language's behavior graph.
- Fahmy and Roos' algorithm exploits the fact that the counter can be changed independently of the input. This does no longer hold for VCAs. Therefore, we can use the counter only in a limited way.

Nevertheless, let us illustrate the difficulties of directly learning 1-VCA-acceptable languages since they reveal interesting details about the problem.

Lemma 5.10. *There is an 1-VCA-acceptable language L for which every configuration graph $G_{\mathcal{A}}$ of an 1-VCA $\mathcal{A}$ recognizing L is not isomorphic to the behavior graph BG_L.*

Proof. Consider the language $L = \{c_1^n r_1^n \mid n > 0\} \cup \{c_2^n r_2^n \mid n > 0\}$ over the alphabet $\Sigma_c = \{c_1, c_2\}$, $\Sigma_r = \{r_1, r_2\}$ and $\Sigma_{int} = \emptyset$. Figure 5.4 shows an 1-VCA, which recognizes L.

Let us first informally introduce the idea of the proof. We observe that all words $w = c_1^n r_1^n$ and $w' = c_2^n r_2^n$ are L-equivalent for every $n > 0$ while $c_1^l \not\sim_L c_2^l$

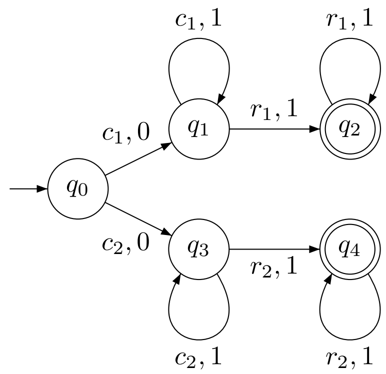


Figure 5.4: A partial 1-VCA recognizing L. Assuming that the VCA has threshold m, a transition of the form $q \xrightarrow{a,i} q'$ where $a \in \Sigma$ and $i \in \{0, \ldots, m\}$ represents the transition $\delta_i(q, a) = q'$

and $c_1^n r_1^l \not\sim_L c_2^n r_2^l$ for every $n > 0$ and $0 \leq l < n$. That means that during the run of an 1-VCA on words w and w' starting with c_1 and c_2 respectively the automaton has to be in different states. Additionally, to produce a configuration graph isomorphic to the behavior graph of L, the last states of each run have to be the same. To archive this, the automaton has to know when it has read $c_1^n r_1^{n-1}$ or $c_2^n r_2^{n-1}$, i.e. when the counter has reached the value 1, and then switch to the same state while reading the last return symbol. Since the behavior graph has infinitely many states, the automaton cannot store this information only in its states and has to access its counter. However, any 1-VCA can only check if the counter is 0 or greater than 0, which makes it impossible to produce a configuration graph isomorphic to the behavior graph of L.

This situation is depicted in Figure 5.5. Both graphs differ only on the gray shaded nodes. To produce isomorphic graphs, the automaton has to know when it has reached the counter value 1 after reading all calls.

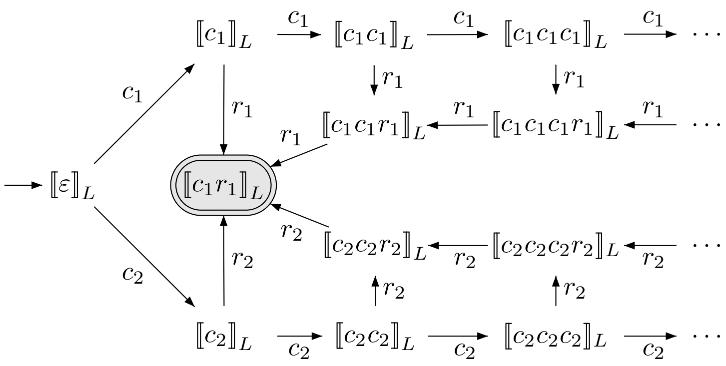
After giving this intuition, we prove the lemma formally. Therefore, let $\mathcal{A} = (Q, \Sigma, q_0, \delta_0, \delta_1, F)$ be an arbitrary 1-VCA recognizing L that has a configuration graph $G_\mathcal{A}$ isomorphic to the behavior graph BG_L and $n = |Q|$ the number of states of $\mathcal{A}$. We consider the words $w = c_1^m r_1^m$ and $w' = c_2^m r_2^m$ where $m > n^2$ and their runs

$$(q_0, 0) \xrightarrow{c_1} (q_1, 1) \xrightarrow{c_1} \ldots \xrightarrow{r_1} (q_{2m-1}, 1) \xrightarrow{r_1} (q, 0)$$

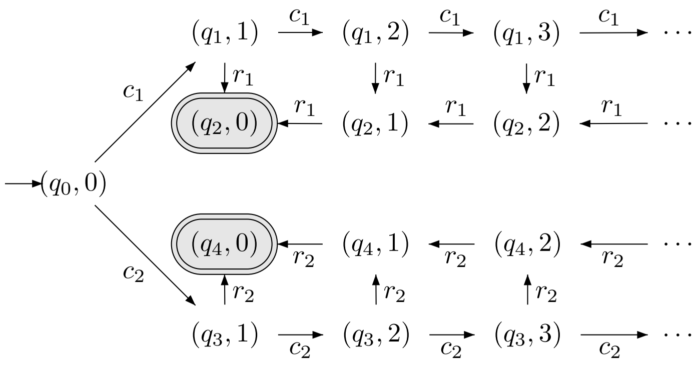
and

$$(q_0, 0) \xrightarrow{c_2} (q_1', 1) \xrightarrow{c_2} \ldots \xrightarrow{r_2} (q_{2m-1}', 1) \xrightarrow{r_2} (q, 0)$$

respectively. Because $\delta_1(q_{2m-1}, r_1) = \delta_1(q_{2m-1}', r_2) = q$, the pair (q_{2m-1}, q_{2m-1}') can only occur once after reading c_1^m and c_2^m. To see this, assume $q_l = q_{2m-1}$ and $q_l' = q_{2m-1}'$ for some $l \in \{m, \ldots, 2m-2\}$. We know that after reading both $c_1^m r_1^{l+1}$ and $c_2^m r_2^{l+1}$, $\mathcal{A}$ is in state q with counter value $m - l - 1$. That means that $c_1^m r_1^{l+1}$ is equivalent to $c_2^m r_2^{l+1}$, which is a contradiction to the above observation. By using the same argument, the pair (q_{2m-2}, q_{2m-2}') can also only occur once after reading c_1^m and c_2^m. This argumentation can be done repetitively and shows that each pair (q_i, q_i') can only occur once after reading c_1^m and c_2^m for each $i \in \{m, \ldots, 2m-1\}$.

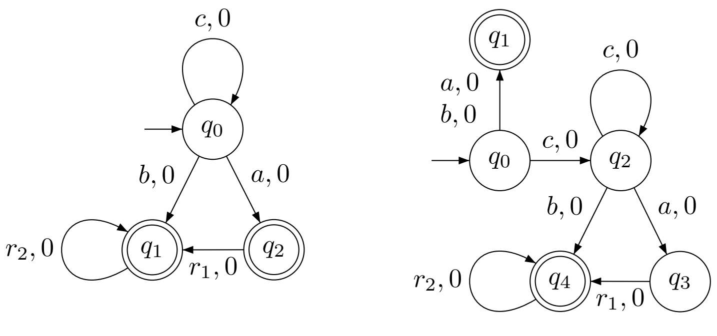


(a) The behavior graph of L



(b) The configuration graph of the 1-VCA in Figure 5.4

Figure 5.5: The behavior graph and configuration graph used in the proof of Lemma 5.10.



(a) The 0-VCA $\mathcal{A}_{min}$

(b) The 0-VCA $\mathcal{A}_{iso}$

Figure 5.6: Two 0-VCAs recognizing the language L from Lemma 5.11

It is clear that there are n^2 pairwise different pairs of states of $\mathcal{A}$. But we showed that there have to be at least $2m - m = m > n^2$ pairwise different pairs of states, which yields a contradiction.

□

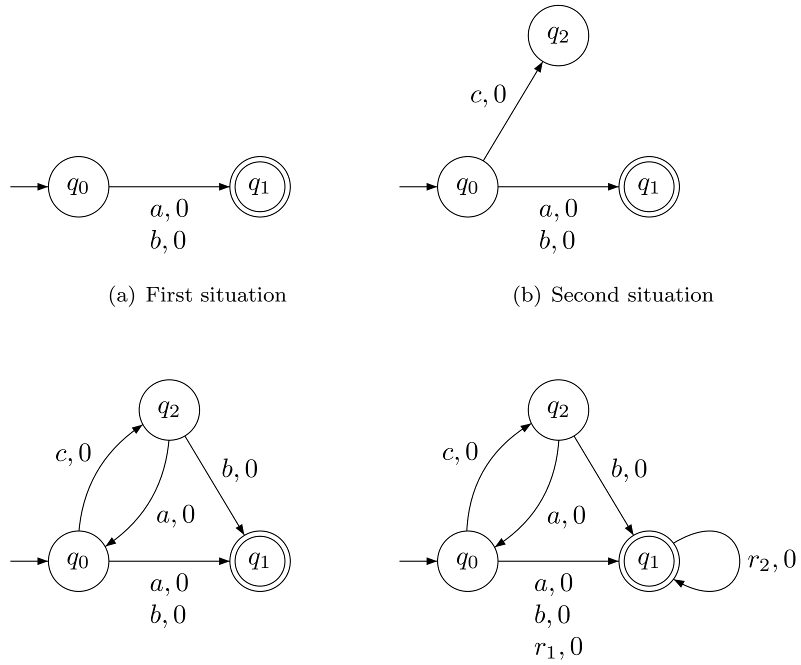
Because of this difficulty, a direct application of Fahmy and Roos' algorithm is not possible. We, therefore, decided to concentrate on a special case, namely the learning of 0-VCA-acceptable languages. On the one hand, we hoped that this would reduce the complexity of the learning task since 0-VCAs do only have one transition function δ_0. On the other hand, we supposed that we could adapt techniques for learning 0-VCAs to the task of learning 1-VCAs. Unfortunately, similarly to the case of 1-VCAs difficulties occurred. It is not hard to verify that Lemma 5.10 also holds for 0-VCAs since the language used in the proof of this lemma is in fact 0-VCA-acceptable. Moreover, we were able to prove that, even if there is a 0-VCA that has an isomorphic configuration graph, this is not necessarily the smallest 0-VCA accepting the language. This shows the absence of a simple canonical object, which could be learned. The next lemma formally shows the latter observation.

Lemma 5.11. *There is a 0-VCA-acceptable language L such that*

- *each minimal 0-VCA accepting L has a configuration graph that is not isomorphic to the behavior graph of L but*
- *there is a 0-VCA accepting L that has a configuration graph isomorphic to the behavior graph of L.*

Proof of Lemma 5.11. Consider the language $L = \{a, b\} \cup \{c^n a r_1 r_2^{n-1} \mid n \geq 1\} \cup \{c^n b r_2^n \mid n \geq 1\}$ over the alphabets $\Sigma_c = \{c\}$, $\Sigma_r = \{r_1, r_2\}$ and $\Sigma_{int} = \{a, b\}$.

A minimal 0-VCA $\mathcal{A}_{min}$ recognizing L is depicted in Figure 5.6(a). We observe that $a \sim_L b$. However, since $\delta_0(q_0, a) \neq \delta_0(q_0, b)$, the configuration graph $G_{\mathcal{A}_{min}}$ is not isomorphic to the behavior graph G_L. Furthermore, it is easy to see that the configuration graph of the 0-VCA $\mathcal{A}_{iso}$ is isomorphic to the behavior graph G_L. The 0-VCA $\mathcal{A}_{iso}$ is shown in Figure 5.6(b).



(a) First situation

(b) Second situation

(c) Third situation

(d) Fourth situation

Figure 5.7: The situations constructed in the proof of Lemma 5.11

At first, we show that there is no 0-VCA with less states than $\mathcal{A}_{min}$ which recognizes L.

- There is no 0-VCA recognizing L with a single state only since $L \neq L_{wm}$ and $L \neq \emptyset$.
- Assume that there is a 0-VCA $\mathcal{A} = (\{q_0, q_1\}, \Sigma, q_0, \delta_0, F)$ with two states recognizing L. Because $\varepsilon \notin L$ and $a, b \in L$, we know that $q_0 \notin F$, $q_1 \in F$ and $\delta_0(q_0, a) = \delta_0(q_0, b) = q_1$. Furthermore, $\delta_0(q_0, c) = q_1$ because $\delta_0(q_0, c) = q_0$ implies $ca \sim_L cb$, which is a contradiction. Since $ca \not\sim_L cb$, we know that $\delta_0(q_1, a) \neq \delta_0(q_1, b)$. Then, either $aa \in L$ or $ab \in L$ holds. This yields a contradiction and proves that there is no 0-VCA with two states which recognizes L.

We now show that there is no 0-VCA with three states that recognizes L and whose configuration graph is isomorphic to the behavior graph of L. We prove this by contradiction. Therefore, assume that the 0-VCA $\mathcal{A} = (\{q_0, q_1, q_2\}, \Sigma, q_0, \delta_0, F)$ recognizes L. Furthermore, we assume that $G_{\mathcal{A}} \cong G_L$.

Since $\varepsilon \notin L$, we know $q_0 \notin F$. To produce a configuration graph isomorphic to the behavior graph of L and since $a \sim_L b$, w.l.o.g. we assume that $\delta_0(q_0, a) = \delta_0(q_0, b) = q_1$. Additionally, we have $q_1 \in F$ because $a \in L$. This situation is depicted in Figure 5.7(a).

We observe that $\delta_0(q_1, a) \neq q_0$ and $\delta_0(q_1, b) \neq q_0$ because $aaa \notin L$ and $aba \notin L$ respectively. Additionally, $\delta_0(q_1, a) \neq q_1$ and $\delta_0(q_1, b) \neq q_1$ since $aa \notin L$ and $aa \notin L$ respectively.

We know that $\delta_0(q_0, c) \neq q_0$ since $ca \not\sim_L cb$. Additionally, $\delta_0(q_0, c) \neq q_1$ has to hold. Otherwise, to archive $ca \not\sim_L cb$, $\delta_0(q_1, a) \neq \delta_0(q_1, b)$ is required. However, this requires either an a- or a b-transition from q_1 to q_0 or q_1 respectively. This is a contradiction to the above observation. Therefore, we state that $\delta_0(q_0, c) = q_2$. This situation is shown in Figure 5.7(b).

Moreover, we observe that $\delta_0(q_2, a) \neq q_2$ since otherwise $ca \sim_L caa$ holds. With the same argument, we gain $\delta_0(q_2, b) \neq q_2$. Furthermore, we know $\delta_0(q_2, a) \neq \delta_0(q_2, b)$ since $ca \not\sim_L cb$. Therefore, there are only two cases left.

1. Assume $\delta_0(q_2, a) = q_0$ and $\delta_0(q_2, b) = q_1$. This situation is depicted in Figure 5.7(c).

   The state q_0 has to have an outgoing r_1 transitions to archive $car_1 \in L$. We first observe that $\delta_0(q_0, r_1) = q_2$ is not possible since $car_1b \notin L$. Furthermore, $\delta_0(q_0, r_1) = q_0$ is not helpful since $q_0 \notin F$. Thus, we have $\delta_0(q_0, r_1) = q_1$.

   Additionally, we infer that $\delta_0(q_1, r_2) \neq q_0$ because $cbr_2 \in L$ but $q_0 \notin F$. Furthermore, $\delta_0(q_1, r_2) \neq q_2$ because $cbr_2b \notin L$. Therefore, we have $\delta_0(q_1, r_2) = q_1$. This situation is depicted in Figure 5.7(d).

   However, this is a contradiction since $cacar_1r_2 \notin L$. Therefore, the assumption $\delta_0(q_2, a) = q_0$ and $\delta_0(q_2, b) = q_1$ is wrong.

2. Assume $\delta_0(q_2, a) = q_1$ and $\delta_0(q_2, b) = q_0$. We can show analogously to case 1. that this assumption is wrong.

Altogether, the 0-VCA $\mathcal{A}$ does not recognize L which is a contradiction. Thus, there is no minimal 0-VCA recognizing L whose configuration graph is isomorphic to G_L. Furthermore, $\mathcal{A}_{iso}$ is a 0-VCA recognizing L, whose configuration graph is isomorphic to G_L. This proves Lemma 5.11.

□

5.3 Learning VCA-acceptable languages

The learning of VCA-acceptable languages has not been addressed in literature so far and, thus, no learning algorithm is know. However, in this section we present our algorithm for learning VCA-acceptable languages. For an VCA-acceptable target language L, our algorithm efficiently computes an m-VCA $\mathcal{A}$ such that $L(\mathcal{A}) = L$. The runtime of the learner is polynomial in some characteristic parameters of the target language.

The learning framework used in this section is a slight variant of the MAT presented in Chapter 3. It turns out that answering membership and equivalence queries is not sufficient in our setting. Therefore, we slightly improve the teacher and allow questions of the form

"Did we learn an initial part of the unknown language correctly?"

Since the MAT is capable of answering equivalence queries, we state that this additional capability is a natural one and does not restrict the area of application. However, to formalize this new type of question, we first need some notations we introduce now.

Let us first briefly recall and introduce some notations of alphabets we use in this section. Let $\hat{\Sigma}$ be a pushdown alphabet consisting of the alphabets Σ_c, Σ_r and Σ_{int} and $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$.

- The set $\Sigma^*_{\geq 0}$ contains all words that have only prefixes of non-negative counter value, i.e. $\Sigma^*_{\geq 0} = \{w \in \Sigma^* \mid \forall u \in pref(w) :\ cv(u) \geq 0\}$. One can think of words of the set $\Sigma^*_{\geq 0}$ as all those words that can be processed by a VCA.
- The set $\Sigma^*_{0,t}$ contains all words that have only prefixes of non negative counter value and do not exceed the value t. Formally, we define $\Sigma^*_{0,t} = \{w \in \Sigma^* \mid \forall u \in pref(w) :\ 0 \leq cv(u) \leq t\}$. These words are especially useful when learning $\sim_L$ up to counter value t.
- The set $\Sigma^*_{=i,t}$ contains all words that have counter value i and whose prefixes have a counter values between 0 and t. The set $\Sigma^*_{=i,t}$ is formally defined as $\Sigma^*_{=i,t} = \{w \in \Sigma^* \mid cv(w) = i\} \cap \Sigma^*_{0,t}$.

5.3.1 The learner

Our learning algorithm is based upon results of Bárány, Löding and Serre [BLS06]. In their paper they introduced a refinement of the Nerode congruence $\sim_L$ over the words of $\Sigma_{\geq 0}$. This new congruence groups all words with respect to $\sim_L$ and their counter value. Formally they define the refined Nerode congruence $\sim'_L \subseteq \Sigma_{\geq 0} \times \Sigma_{\geq 0}$ as

$$u \sim'_L v \ \Leftrightarrow\ cv(u) = cv(v) \text{ and } u \sim_L v$$

for all $u, v \in \Sigma_{\geq 0}$. Since $cv(u) = cv(v)$ holds for all $\sim'_L$-equivalent words u and v, we say that the equivalence class of u has counter value i if $cv(u) = i$ and write $cv([\![u]\!]_L) = i$.

In fact, because of the visibly property, we know that two $\sim_L$-equivalent words u and v have an equal counter value if they can be extended to a word in L. Thus, $\sim'_L$ only refines the single equivalence class that contains all words that cannot be extended to a word in L. For the rest of this section we exclusively use the refined Nerode congruence $\sim'_L$. Note that the use of the refined Nerode congruence also slightly changes the definitions relying on it such as behavior graphs and equivalence classes. However, for reasons of simplicity we use the same notations as for $\sim_L$ in this section.

As the algorithm of Fahmy and Roos, our learning algorithm learns an initial fragment of the behavior graph, this time w.r.t. the refined Nerode congruence, and decomposes it into an m-VCA. To construct an m-VCA from an initial fragment of BG_L, we use some facts about the behavior graph, which we now describe in detail.

We first observe that the number of equivalence classes of $\sim_L$ on the same *level*, i.e. all equivalence classes that have the same counter value, is bounded by a constant K. In fact, we can bound K by the number of states of a VCA accepting L since each VCA can only use its state to distinguish non-equivalent words with the same counter value. Behavior graphs that have this special property are called *K-slender* and the value K is called the *slenderness index*.

The fact that BG_L is K-slender allows us to enumerate the equivalence classes on a level. Formally, this can be done by a mapping

$$\nu : Q_L \to \{1, \ldots, K\}$$

that assigns a number between 1 and K to each equivalence class. We assume that ν assigns different numbers to all classes on the same level and that this numbering is done in ascending order.

With this enumeration ν the behavior graph can be coded as a sequence of mappings τ_i, $i = 1, \ldots$ where τ_i codes the edges of the equivalence classes of the i-th level, i.e. the vertex set $\{[\![u]\!]_L \mid u \in \Sigma_{\geq 0}$ and $cv(u) = i\}$. The (partial) mapping τ_i assigns to each pair (j, a) of an equivalence class number and an input symbol the number of the equivalence class that is reached from class number j on level i on reading a. If there is no equivalence numbered with j on this level, then the mapping is undefined. So the edges of the i-th level are fully described by τ_i. More formally, the description $\tau_i : \{1, \ldots, K\} \times \Sigma \to \{1, \ldots, K\}$ for $i = 1, \ldots$ of the i-th level of BG_L is defined by

$$\tau_i(j, a) = j'$$

if there is an equivalence class $[\![u]\!]_L$ on the i-th level with $\nu([\![u]\!]_L) = j$ and $\nu([\![ua]\!]_L) = j'$. Additionally, $\tau_i(j, a)$ is undefined if there is no equivalence class with $\nu([\![u]\!]_L) = j$. It is clear that the infinite word

$$\alpha = \tau_0 \tau_1 \ldots$$

fully describes the behavior graph of L. Thus, we call α a *description* of BG_L. Since a language uniquely determines its behavior graph, we also say that α is a description of L.

By using the description of a behavior graph, Bárány, Löding and Serre were able to show that the behavior graph of a VCA-acceptable language has a repeating structure. Precisely, they showed the following theorem.

Theorem 5.12 (Bárány, Löding and Serre [BLS06])**.** *Let L be a VCA-acceptable language. Then, there is an enumeration $\nu : Q_L \to \{1, \ldots, K\}$ such that the corresponding description α of the behavior graph BG_L is an ultimately periodic word with offset m and period k, i.e.*

$$\alpha = \tau_0 \ldots \tau_{m-1} \left(\tau_m \ldots \tau_{m+k-1}\right)^\omega \quad .$$

Note that there are infinitely many different offsets and periods for an ultimately periodic description of the behavior graph. For instance we simply obtain a new offset or a new period by adding a multiple of k. However, each VCA-acceptable language has an (up to a relabeling of the equivalence classes) unique *characteristic description*, namely the one with the smallest offset and the smallest period for this offset.

The next Construction shows that an m-VCA that knows whether it is in the offset part of an ultimately periodic description (by using its threshold) or in the periodic part (by using a modulo-k-counter to keep track of the position within the period) can simulate BG_L. Bárány, Löding and Serre provide a construction for an appropriate m-VCA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.

Construction 5.13 (Bárány, Löding and Serre [BLS06]). *Let L be a VCA-acceptable language and $\alpha = \tau_0 \dots \tau_m \, (\tau_{m+1} \dots \tau_{m+k})^\omega$ an ultimately periodic description of the behavior graph BG_L with offset m and period k. The m-VCA $\mathcal{A}_\alpha = (Q, \Sigma, q_0, \delta_0, \dots, \delta_m, F)$ is defined as*

- $Q = \{1, \dots, K\} \times \{0, \dots, k-1\}$,
- $q_0 = (\nu(\llbracket \varepsilon \rrbracket_L), k-1)$,
- $F = \{\nu(\llbracket u \rrbracket_L) \mid u \in L\} \times \{0, \dots, k-1\}$ *and*
- *the transitions functions $\delta_0, \dots, \delta_m$ defined by:*
  - *For every $j < m$, $i \in \{1, \dots, K\}$ and $0 \leq r < k$. If $j = m-1$ and $a \in \Sigma_c$ let*
  $$\delta_{m-1}((i, r), a) = (\tau_{m-1}(i, a), 0)$$
  and if $j < m-1$ or $a \notin \Sigma_c$ let
  $$\delta_j((i, r), a) = (\tau_j(i, a), k-1) \ .$$
  - *For every $a \in \Sigma$, $i \in \{1, \dots, K\}$ and $0 \leq r < k$ let*
  $$\delta_m((i, r), a) = (\tau_{m+r}(i, a), (r + \chi(a)) \mod k) \ .$$

The fundamental idea of their construction is to exploit the fact that, because the periodic part of α repeats ad infinitum, the language is completely described by a finite prefix $\tau_0 \dots \tau_{m-1} \tau_m \dots \tau_{m+k-1}$ of length $m + k$ of α. Moreover, since the behavior graph is K-slender, it can be simulated by a VCA, whose states consists of two components: During this simulation, the first component stores the current number of the equivalence class. The second component keeps track of the position in the period if the VCA is currently simulating the behavior graph in its periodic part. The transitions are defined according to the description α.

Bárány, Löding and Serre did not provide an explicit proof of correctness. We catch up with this in the following lemma.

Lemma 5.14. *Let L be a VCA-acceptable language and $\mathcal{A}_\alpha$ the m-VCA constructed in Construction 5.13. Then,*

$$((\nu(\llbracket \varepsilon \rrbracket_L), k-1), 0) \xrightarrow{w}_{\mathcal{A}_\alpha} \begin{cases} ((\nu(\llbracket w \rrbracket_L), k-1), cv(w)) \\ \text{, if } cv(w) < m \\ \\ ((\nu(\llbracket w \rrbracket_L), (cv(w) - m) \mod k), cv(w)) \\ \text{, if } cv(w) \geq m \end{cases}$$

holds for all $w \in \Sigma^*_{\geq 0}$.

The proof is a straightforward but rather technical induction. In the induction step, we have to make a distinction depending on the counter value and the type of the next input symbol to treat each individual cases of the definition of the transition functions properly. However, the situation when $\mathcal{A}_\alpha$ is simulating the i-th level in the periodic part is more interesting. The fact that the description α is ultimately periodic allows us to guarantee that the simulation is correct even if $\tau_{m+((i-m) \mod k)}$ is used instead of τ_i. This clearly shows that, because of the repeating structure of BG_L, the finite information about the offset part and the first periodic part is sufficient to properly simulate the behavior graph of L.

Proof of Lemma 5.14. We proof Lemma 5.14 by induction over the length of an input $w \in \Sigma^*_{\geq 0}$.

Let $w = \varepsilon$. Then, the lemma clearly holds by definition of runs of m-VCAs.

Let $w = ua$. Since $w \in \Sigma_{\geq 0}$, we also know that $u \in \Sigma_{\geq 0}$. We distinguish the following cases.

- Let $cv(u) = j < m$. Then, we can apply the induction hypothesis for u and gain
$$((\nu(\llbracket \varepsilon \rrbracket_L), k-1), 0) \xrightarrow{u}_{\mathcal{A}_\alpha} ((\nu(\llbracket u \rrbracket_L), k-1), j)\ .$$
  - If $j = m-1$ and $a \in \Sigma_c$, then we know from the definition of δ_{m-1} that
$$\delta_{m-1}((\nu(\llbracket u \rrbracket_L), k-1), a) = (\underbrace{\tau_{m-1}(\nu(\llbracket u \rrbracket_L), a)}_{=\nu(\llbracket ua \rrbracket_L)}, 0)$$
    and, thus,
$$((\nu(\llbracket u \rrbracket_L), k-1), m-1) \xrightarrow{a}_{\mathcal{A}_\alpha} ((\nu(\llbracket ua \rrbracket_L), 0), m).$$
    Since $(m-m) \mod k = 0$ and with the result obtained from applying the induction hypothesis, we deduce that the lemma holds.
  - If $j < m-1$ or $a \notin \Sigma_c$, then we know from the definition of δ_j that
$$\delta_j((\nu(\llbracket u \rrbracket_L), k-1), a) = (\tau_j(\nu(\llbracket u \rrbracket_L), a), k-1)$$
    and, hence,
$$((\nu(\llbracket u \rrbracket_L), k-1), j) \xrightarrow{a}_{\mathcal{A}_\alpha} ((\nu(\llbracket ua \rrbracket_L), k-1), j + \chi(a)).$$
    Again, since $cv(ua) = j + \chi(a)$ and considering the result obtained from applying the induction hypothesis, we infer that the lemma holds.
- Let $cv(u) = j > m$. Then, we know
$$((\nu(\llbracket \varepsilon \rrbracket_L), k-1), 0) \xrightarrow{u}_{\mathcal{A}_\alpha} ((\nu(\llbracket u \rrbracket_L), (j-m) \mod k), j)$$
  from applying the induction hypothesis.

  Let $r = (j-m) \mod k$. Since α is an ultimately periodic word with offset m and period k, we know that $\tau_{m+r} = \tau_j$ and, hence,
$$\nu(\llbracket ua \rrbracket_L) = \tau_j(\nu(\llbracket u \rrbracket_L), a) = \tau_{m+r}(\nu(\llbracket u \rrbracket_L), a)\ .$$
  In other words, we can use τ_{m+r} instead of τ_j to properly simulate the run of BG_L. Thus, the definition
$$\delta_m((\nu(\llbracket u \rrbracket_L), r), a) = (\tau_{m+r}(\nu(\llbracket u \rrbracket_L), a), (r + \chi(a)) \mod k)$$

is correct and yields

$$((\nu(\llbracket u \rrbracket_L), r), j) \xrightarrow{a}_{\mathcal{A}_\alpha} ((\nu(\llbracket ua \rrbracket_L), (r + \chi(a)) \mod k), j + \chi(a)) \ .$$

Since $(r + \chi(a)) \mod k = (cv(w) - m) \mod k$, $j + \chi(a) = cv(w)$ and considering the result obtained from applying the induction hypothesis, we infer that the lemma also holds for this case.

□

Note that Construction 5.13 slightly differs from the one given by Bárány, Löding and Serre. Originally, they used the $(\_, 0)$ component of $\mathcal{A}_\alpha$ to simulate the behavior graph in its offset part instead of the $(\_, k-1)$ component as we do. However, this does not make a difference if the transitions are adapted accordingly.

Construction 5.13 and its proof of correctness reveal some facts, which we exploit in the construction of our learning algorithm.

1. It is not hard to verify that the m-VCA $\mathcal{A}_\alpha$ induces a configuration graph $G_{\mathcal{A}_\alpha}$, which is isomorphic to the behavior graph BG_L. Thus, we can use the behavior graph to decompose it into a VCA accepting the language L (the m-VCA $\mathcal{A}_\alpha$).

2. The construction and its proof of correctness show that a finite prefix of an ultimately periodic description of L and, hence, a finite initial fragment of BG_L is sufficient to construct a VCA recognizing L. In fact, we only need to know BG_L up to level $m + k - 1$ to be able to construct $\mathcal{A}_\alpha$ since this information already describes the whole behavior graph.

The second point states that it is sufficient to learn an initial fragment of the behavior graph BG_L up to level $m + k - 1$. However, since we do not know the offset m and the period k in advance, we need to extract this information from our learned initial fragment. It is clear that the level $m+k-1$ is too small for this purpose because it does not allow to verify that the part from level m to level $m + k - 1$ is repeating. To do so, we need to learn BG_L at least up to level $m + 2k - 1$.

The learning of the initial fragment of BG_L is done incrementally. We learn an initial fragment of BG_L up to a level t starting with $t = 0$ and then increment t until the value $m + 2k - 1$ is reached or exceeded. However, it turns out that learning all equivalence classes $\{\llbracket u \rrbracket_L \mid u \in \Sigma_{\geq 0} \text{ and } cv(u) = i\}$ with $i \leq t$ for a small t is hard to achieve since it requires the consideration of words that possibly have a counter value higher than t. To be able to handle those words, we are forced to increment t. To avoid this, we define an initial fragment of a behavior graph up to level t slightly different as the part of the behavior graph that can be reached by words of $\Sigma^*_{0,t}$. Like this we can ensure that it is possible to completely learn this initial fragment without having to increment t. A formal definition of this initial fragment is given next. Figure 5.8 shows the situations mentioned above.

Definition 5.15. [Initial fragment of a behavior graph up to level t] Let L be a VCA-acceptable language and $t \geq 0$. The *initial fragment of a behavior graph up to level* t is a tuple $BG_L|_t = (Q_L|_t, q_0^L, \delta_L|_t, F_L|_t)$ where

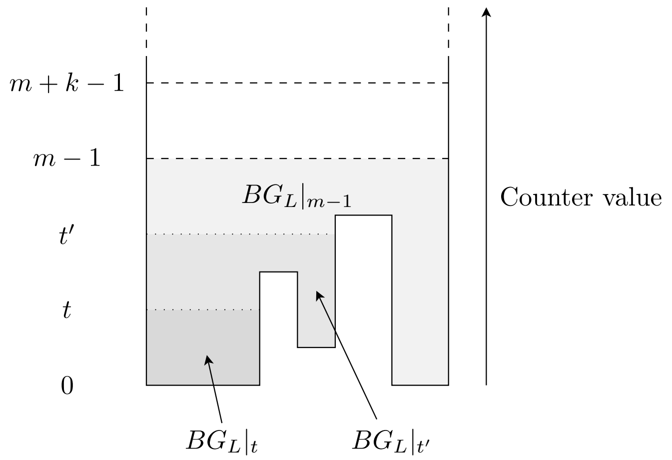


Figure 5.8: A schematic view of an initial fragment of a VCA-acceptable language's behavior graph whose characteristic description has offset m and period k. The figure also shows the initial fragments for the levels t, t' and $m-1$. It is clear that $BG_L|_{t'}$ contains $BG_L|_t$ and $BG_L|_{m-1}$ contains both $BG_L|_t$ and $BG_L|_{t'}$

- $Q_L|_t = \{[\![u]\!]_L \mid u \in \Sigma^*_{0,t}\}$,
- $q_0^L = [\![\varepsilon]\!]_L$,
- $F_L|_t = \{[\![u]\!]_L \mid u \in \Sigma^*_{0,t} \text{ and } u \in L\}$ and
- $\delta_L|_t$ defined by
$$\delta_L|_t([\![u]\!]_L, a) = [\![ua]\!]_L$$
for all $[\![u]\!]_L, [\![ua]\!]_L \in Q_L|_t$ and $a \in \Sigma$. Moreover, $\delta_L|_t([\![u]\!]_L, a)$ is undefined if $[\![ua]\!]_L \notin Q_L|_t$.

$\lhd$

One can think of $BG_L|_t$ as the behavior graph of a restriction of the refined Nerode congruence defined by

$$\sim_L |_t = \sim_L \cap \left(\Sigma^*_{0,t} \times \Sigma^*_{0,t}\right) .$$

Note that the number of equivalence classes of $\sim_L |_t$ is bounded by K and t. To be precise, it is not hard to see that $index(\sim_L |_t) \leq K \cdot t$.

As one can see in Figure 5.8, the behavior graph $BG_L|_t$ may not contain all L-equivalence classes that have counter value t or less. It is clear that a complete initial fragment of the behavior graph is needed in order to compute a VCA accepting the language L. It is, therefore, substantial that we eventually know the behavior graph in its whole "width". However, it is not hard to verify that this happens automatically when the level reaches the value $m+k-1$. We state this important fact in the following observation.

Observation 5.16. If $t \geq m + k - 1$ then $\sim_L |_t$ contains all equivalence classes $\{[\![u]\!]_L \mid u \in \Sigma^*_{\geq 0} \text{ and } cv(u) \leq t\}$.

$\lhd$

The stratified observation table

To learn an initial fragment of the behavior graph, our algorithm uses an adaptation of the observation table presented by Angluin in her algorithm for learning regular languages [Ang87]. Analogous to Angluin, we do not manage the behavior graph in an explicit way but learn an approximation of the restricted refined Nerode congruence $\sim_L |_t$. The learning of this approximation starts with a coarse partition of all words of $\Sigma^*_{0,t}$ and refines the approximation until $\sim_L |_t$ is eventually computed.

The refined Nerode congruence provides a view on the behavior graph, which splits it up w.r.t. to the counter value of the equivalence classes. Therefore, we have to respect this in the design of our observation table. Moreover, we now have to deal with three different types of input symbols, namely call, return and internal symbols. A formal definition of a *stratified observation table* as we call it can be found in Definition 5.17.

Definition 5.17. [Stratified observation table] A *stratified observation table up to level* $t \geq 0$ is a tuple $O = ((R_i)_{i=0,\dots,t}, (S_i)_{i=1,\dots,t}, T)$ consisting of

- nonempty, finite sets $R_i \subseteq \Sigma^*_{=i,t}$ of *representatives* for all $i = 0, \dots, t$,
- nonempty, finite sets $S_i \subseteq \Sigma^*$ of *samples* for all $i = 0, \dots, t$ and
- a mapping
  $$T : \ (D_{int} \cup D_c \cup D_r) \to \{0, 1\}$$
  where
  $$\begin{aligned} D_{int} &= \bigcup_{i=0}^{t} (R_i \cup R_i \cdot \Sigma_{int}) \cdot S_i \ , \\ D_c &= \bigcup_{i=0}^{t-1} R_i \cdot \Sigma_c \cdot S_{i+1} \ , \\ D_r &= \bigcup_{i=1}^{t} R_i \cdot \Sigma_r \cdot S_{i-1} \ . \end{aligned}$$

Thereby, we require that $uw \in \Sigma^*_{\geq 0}$ and $cv(uw) = 0$ holds for all $u \in R_i$, $w \in S_i$ and $i = 0, \dots, t$. We call O a *stratified observation table for* L if and only if

$$\forall w \in (D_{int} \cup D_c \cup D_r) : \ T(w) = 1 \Leftrightarrow w \in L$$

holds. Moreover,

$$R = \bigcup_{i=0}^{t} R_i \ \text{ and } \ S = \bigcup_{i=0}^{t} S_i$$

are the sets of *representatives* and *samples*.

◁

In the following we assume (and ensure this by defining the learning algorithm accordingly) that O is always a stratified observation table for the target language L.

Intuitively, a stratified observation table manages approximations of the equivalence classes of $\sim_L |_t$ separately for each level. Each set R_i contains representatives with counter value i for the equivalence classes of the i-th level. The set S_i contains samples to distinguish the representatives of the i-th level. Thereby, we require that S_i only contains words that are "valid", i.e. that fulfill the condition $uw \in \Sigma^*_{\geq 0}$ and $cv(uw) = 0$ for all $u \in R_i$. The actual information of a stratified observation table is stored in the mapping T. One can think of T as three different tables for each level where the rows are labeled with representatives and the columns are labeled with samples: The first table of level i stores information about the representatives of the current level and their internal action successors, while the second table stores information about call successors. Finally, the third stores information about return successors. However, note that there are no call successors for representatives of level t (since we only want to learn the behavior graph up to level t) and no return successors for representatives on level 0 (since such words cannot be processed by a VCA).

A stratified observation table induces an equivalence on the set $\Sigma^*_{0,t}$ in a similar manner as the refined Nerode congruence.

Definition 5.18. Let O be a stratified observation table for L. Two words $u, v \in \Sigma^*_{0,t}$ are said to be *O-equivalent*, denoted by $u \sim_O v$, if and only if

$$cv(u) = cv(v) \;\text{ and }\; \forall w \in S_{cv(u)} : \; uw \in L \Leftrightarrow vw \in L$$

holds.

$\lhd$

Besides the fact that this definition of O-equivalence also takes the counter value of words into consideration, it differs from the original one given in Chapter 3 (cf. Definition 3.3) on two fundamental facts.

- First, the O-equivalence is not restricted to representatives of the set R but defined for all words $u, v \in \Sigma^*_{0,t}$. This is needed to properly define the extension of the MAT.

- Second, the O-equivalence is not defined by means of the data stored in the observation table. However, since O is a stratified observation table for L, the data stored in the table agrees with L. This means that two O-equivalent representatives have the same row.

For a word $u \in \Sigma^*_{0,t}$ we define its O-equivalence class as $[\![u]\!]_O = \{v \in \Sigma^*_{0,t} \mid u \sim_O v\}$. The value of $index(\sim_O)$ is defined slightly different as the number of O-equivalence classes $[\![u]\!]_O$ for which there is a representative $u \in R$. In other words, $index(\sim_O)$ counts the equivalence classes contained in the table. Note that the total number of O-equivalence classes can be higher. However, since $u \sim_L v$ implies $u \sim_O v$, the total number of O-equivalence classes, and thus also $index(\sim_O)$, is bounded by $index(\sim_L)$.

It can be the case that the information stored in a stratified observation table is incomplete in the sense that it does not represent a congruence relation. In this case, we have not yet gathered enough information about the target language. Therefore, we define the following two conditions.

- We call a stratified observation table *closed* if and only if the condition
$$\forall u \in R_i \ \forall a \in \Sigma : \ [\![ua]\!]_O \cap R_{i+\chi(a)} \neq \emptyset$$
holds for all $i = 0, \ldots, t$ except for $i = 0$ and $a \in \Sigma_r$ as well as $i = t$ and $a \in \Sigma_c$.

- We call a stratified observation table *consistent* if and only if the condition
$$\forall u, v \in R_i \ \forall a \in \Sigma : \ u \sim_O v \Rightarrow ua \sim_O va$$
holds for all $i = 0, \ldots, t$ except for $i = 0$ and $a \in \Sigma_r$ as well as $i = t$ and $a \in \Sigma_c$.

Intuitively, a table that is not closed contains a representative $u \in R_i$ such that the equivalence class $[\![ua]\!]_O$ of its a-successor is not present in the table. Note that we need to search for equivalent representatives as well in R_i as in R_{i+1} and in R_{i-1} depending on the type of the symbol a. If we discover O not to be closed, we can easily solve this problem: We know that there is a $u \in R_i$ and an $a \in \Sigma$ such that $[\![ua]\!]_L \cap R_{i+\chi(a)} = \emptyset$. We add ua to $R_{i+\chi(a)}$ and complete the table by asking membership queries for new entries of the row ua. Thus, the closed condition is now satisfied for the representative u. Moreover, $index(\sim_O)$ increases.

The consistent condition expresses the fact that we have not yet found a sample to distinguish two non L-equivalent but O-equivalent representatives $u, v \in R$. This is indicated by the fact that the a-successors of u and v are not O-equivalent for some $a \in \Sigma$. In this case, we know that there is a $w \in S_{cv(u)+\chi(a)}$ such that $uaw \in L \Leftrightarrow vaw \notin L$, i.e. w is the witness that $ua \not\sim_O va$. To separate u and v, we add aw to $S_{cv(u)}$ and update O for the new entries of row aw. Thus, the consistent condition is now satisfied for the words u and v. Moreover, we observe that, since u and v are no longer O-equivalent, both the number of O-equivalence classes and $index(\sim_O)$ increases.

Let us briefly summarize these observations.

Observation 5.19. Let L be a VCA-acceptable language and O a stratified observation table up to a fixed level t. Then, $index(\sim_O) \leq index(\sim_L) = K \cdot t$ where K is the slenderness index of BG_L. Moreover, since $index(\sim_O)$ increases every time we extend the table while it is not closed or not consistent, a stratified observation table can be not closed or not consistent at most $K \cdot t$ times.

$\triangleleft$

If a stratified observation table stores is closed and consistent, we can use it to construct a DFA. Its states are the equivalence classes contained in the table and its transitions are defined according to the congruence relation. Formally, for a stratified observation table O we define the (partial DFA) $\mathcal{A}_O$ as follows.

Construction 5.20. *Let O be a closed and consistent observation table. The DFA $\mathcal{A}_O = (Q_O, \Sigma, q_0^O, \delta_O, F_O)$ is defined as*

- $Q_O = \{[\![u]\!]_O \mid u \in R\}$,
- $q_0^O = [\![\varepsilon]\!]_O$,
- $F_O = \{[\![u]\!]_O \in Q_O \mid u \in L\}$ *and*

- δ_O *defined by*

$$\delta_O([\![u]\!]_O, a) = \begin{cases} [\![ua]\!]_O & \text{, if } [\![ua]\!]_O \in Q_O \\ \text{undefined} & \text{, else} \end{cases}$$

The DFA $\mathcal{A}_O$ represents the behavior graph of the approximation of $\sim_L|_t$, which we have learned so far. It is clear that we cannot guarantee this initial fragment to be correct. To check this, we need a new type of query.

Extension of the minimally adequate teacher

It is obvious that computing a VCA from the information stored in a stratified observation table can only result an equivalent VCA if $\mathcal{A}_O$ is not just an approximation of $\sim_L|_t$ but $\sim_L|_t$ itself. Only in this case, we are able to identify the correct repeating structure of $BG_L|_t$ and construct an m-VCA $\mathcal{A}$ accepting the target language. However, it is not possible to learn $\sim_L|_t$ completely by using only membership and equivalence queries: The counter-examples returned by the MAT can have prefixes of arbitrary, increasing counter value. Such counter-examples provide additional information about the behavior graph on levels greater than t but may not allow to learn $\sim_L|_t$ completely. It is, therefore, necessary to require that the MAT provides "useful" counter-examples in the sense that they allow us to learn $\sim_L|_t$ completely.

Since this is a rather strict requirement, we introduce a new kind of query instead that is easy to answer on the one hand and allows a learner to completely learn $\sim_L|_t$ on the other hand. On this new kind of query, we call it *partial equivalence query*, the MAT is provided with a DFA $\mathcal{A}$ representing the behavior graph of a congruence and has to check whether this DFA is an equivalent, i.e. an isomorphic, representation of $BG_L|_t$ or not. Since our representation of $BG_L|_t$ is the DFA $\mathcal{A}_O$, the states are O-equivalence classes. Thus, we can reformulate this query and require the MAT to check whether

$$\forall u, v \in \Sigma^*_{0,t} : \; u \sim_O v \Leftrightarrow u \sim_L v$$

holds.[3] If this condition is satisfied for all $u, v \in \Sigma^*_{0,t}$, then the MAT replies "yes". Otherwise, there are some $u, v \in \Sigma^*_{0,t}$ such that $u \sim_O v$ but $u \not\sim_L v$ meaning that there is a witness w indicating that u and v are not L-equivalent. Then, the MAT returns both uw and vw. Note that prefixes of the words uw and vw can exceed the current level t of the stratified observation table but $uw, vw \in \Sigma^*_{\geq 0}$ and $cv(uw) = cv(vw) = 0$ always holds.

The learning algorithm

The idea of our learning algorithm is to compute the characteristic description α of the target language and then to construct the automaton $\mathcal{A}_\alpha$. From Construction 5.13 we know that a finite prefix of α of length $m + k$, where m is the offset and k the period of α, is sufficient for this purpose. Since the characteristic description is obtained from the behavior graph BG_L, we learn a sufficient large initial fragment $BG_L|_t$ (at least up to level $t = m + k - 1$) and then search

[3] In fact, it is sufficient for the MAT to check whether $u \sim_O v$ implies $u \sim_L v$ since we know that $u \sim_L v \Rightarrow u \sim_O v$ always holds

for the characteristic description. The initial fragment of the behavior graph is represented by $\mathcal{A}_O$.

However, we have to deal with the problem that we do not know the offset m and the period k in advance and, thus, do not know up to which level we have to learn BG_L. We address this problem by trial and error and learn BG_L gradually, starting from $t = 0$ and then incrementing t.[4] For each increment we ensure that $BG_L|_t$ is learned correctly by asking partial equivalence queries and try to find descriptions α' of $BG_L|_t$ that are periodic. We call such finite descriptions of $BG_L|_t$ *periodic descriptions*. However, since $BG_L|_t$ may contains only a part of the information about L, there can be several periodic descriptions with distinct offsets and periods. In this case, we construct the conjectures $\mathcal{A}_{\alpha'}$ for all of them and ask equivalence queries respectively. If one of them is a sufficient large prefix of the characteristic description, then the MAT returns a positive answer and the learner returns this VCA. If all conjectures are not equivalent to L, then none of the periodic descriptions found are sufficient large prefixes of the characteristic description and, hence, the initial fragment $BG_L|_t$ is not large enough. In this case, we increment the level t and repeat the procedure.

It is not hard to verify that a sufficient large prefix of the characteristic description is found if we learn the initial fragment of BG_L at least up to level $m+k-1$. However, even if this is sufficient to construct an equivalent m-VCA, identifying a periodic description requires that we learn $BG_L|_t$ at least up to level $m + 2k - 1$.

Our learner for VCA-acceptable languages is shown in pseudo code as Algorithm 7. It starts with an initial stratified observation table with threshold $t = 0$ and $R_0 = S_0 = \{\varepsilon\}$. The algorithm consists of a main loop, in which the following two steps are executed consecutively:

1. We learn $\sim_L|_t$ by refining $\sim_O$. The DFA $\mathcal{A}_O$ represents $BG_L|_t$.

2. We compute all periodic descriptions α of $\mathcal{A}_O$ and construct the VCAs $\mathcal{A}_\alpha$. In the case that $\mathcal{A}_O$ does not have a periodic description, we use $\mathcal{A}_O$ as a (partial) VCA. For each of those VCAs we ask an equivalence query. If the MAT returns "yes" to one of these queries, then we return the VCA. If none of these VCAs is equivalent, we uniform randomly choose a counter example and add it as well as all of its prefixes to the representatives of the table. Thereby, the level t of the table is increased.

Let us now describe both steps in detail. The necessary procedure for the first step is shown in Algorithm 8. In this step the task is to learn $\sim_L|_t$. This is done by extending the stratified observation table according to Page 115 until it is both closed and consistent. The function update(O) completes the table after new entries have been added. Then, the DFA $\mathcal{A}_O$ is constructed and a partial equivalence query is asked. Let us assume that we did not learn enough information about L, thus, $\mathcal{A}_O$ is not equivalent to $BG_L|_t$ and the MAT responds with counter-examples u, v. We then know that there is a decomposition of u and v into $u = u'w$ and $v = v'w$ such that $u', v' \in \Sigma^*_{0,t}$, $u' \sim_O v'$ and $w \in \Sigma^*$. To ensure that u' and v' are no longer O-equivalent, it is sufficient to add u' and v' as representatives to the set $R_{cv(u')}$ and w to $S_{cv(u')}$. However, the problem thereby is that we do not know the exact decomposition.

[4]In fact, the increment of t is determined by the counter-example returned by the MAT to an equivalence query

Input: A MAT for the unknown language L
Output: An m-VCA $\mathcal{A}$ with $L(\mathcal{A}) = L$

Create an initial stratified observation table O up to level $t = 0$ with $R_0 = S_0 = \{\varepsilon\}$ and update T;
repeat
```
/* 1. step:                                              */
/* Learning of ~L |t                                     */
```
Refine $\sim_O$ until $\sim_L|_t$ is eventually computed;
```
/* 2. step:                                              */
/* Constructing an equivalent m-VCA from BG_L|t          */
```
Compute all periodic descriptions α of $BG_L|_t$ using $\mathcal{A}_O$;
Construct VCAs for the found periodic descriptions or use $\mathcal{A}_O$ as a partial VCA if no periodic description is found;
if *no equivalent VCA is found* **then**
Add a counter-example to O;
end
until *an equivalent VCA is found* ;
return the found VCA;

Algorithm 7: The learner for VCA-acceptable languages

To overcome this problem, the idea is to first add all decompositions $u = xy$ with increasing length of the prefix x to the table. Thereby, we add x to $R_{cv(x)}$ and y to $S_{cv(x)}$. We stop adding decompositions of u to the table as soon as the first prefix with counter value higher than t is encountered. We proceed with v in the same way. Since $u', v' \in \Sigma^*_{0,t}$, we know that u', v' and w are added properly to the table. This procedure is done by the function `addPartial` shown on page 130.

The loop conditions of the outer loop in Algorithm 8 ensures that this procedure learns $\sim_L|_t$ correctly if it terminates.

Let us now consider the more demanding second step of the learning algorithm. It is shown as Algorithm 9. The first task we have to solve in this step is to compute all periodic descriptions α of $\mathcal{A}_O$. To keep things simple in the beginning, we assume that an oracle tells us all pairs of offset m and period k such that there is a numbering ν that allows to describe $\mathcal{A}_O$ as

$$\alpha = \tau_0 \dots \tau_{m-1} \left(\tau_m \dots \tau_{m+k-1}\right)^i \tau_m \dots \tau_{m+j}$$

where $2 \leq i$, $0 \leq j < k - 1$ and $m + i \cdot k + j = t$. We later show how an appropriate search can replace such an oracle.

However, even if an oracle tells us the offset and period, we still need to find an appropriate numbering. To do so, we split this task in two parts: The numbering of the offset part of $\mathcal{A}_O$ is easy since we can choose an arbitrary numbering. To compute a numbering of the periodic part, we use a *parallel breadth-first search (PBFS)* in the state space of $\mathcal{A}_O$ as sketched in Figure 5.9. This search performs two breadth-first traversals starting from the equivalence classes $[\![u]\!]_O$ on level m and $[\![v]\!]_O$ on level $m + k$ "in parallel" and succeeds if a repeating structure is detected (the precise mechanism of this parallelism is

```
repeat
    while O is not closed or not consistent do
        if O is not consistent then
            Find u ~_O v, a ∈ Σ and w ∈ S_{cv(u)+χ(a)} such that
            T(uaw) = 1 ⇔ T(vaw) ≠ 1;
            S_{cv(u)} := S_{cv(u)} ∪ {aw};
            update(O);
        end
        if O is not closed then
            Find u ∈ R and a ∈ Σ such that ⟦ua⟧_O ∩ R_{cv(u)+χ(a)} = ∅;
            R_{cv(ua)} := R_{cv(ua)} ∪ {ua};
            update(O);
        end
    end
    Compute A_O;
    if the MAT replies with a counter-examples u, v to a partial
    equivalence query with A_O then
        addPartial(O, u, v);
    end
until the MAT replies "yes" to a partial equivalence query with A_O ;
```

Algorithm 8: The first step of the learner for VCA-acceptable languages

unimportant). Both traversals are synchronized in a way that the same action is performed in each step of the breadth-first search. Since we know that the offset is m and the period is k, we restrict both individual searches to the equivalence classes with counter value i where $i \in \{m, \ldots, m + k - 1\}$ and $i \in \{m + k, \ldots, m + 2k - 1\}$ respectively. We call those sets *level ranges* of the traversals. The intuitive idea is that a repeating structure is detected if exactly the same actions of one traversal can be repeated by the other and vice versa.

Each single breadth-first search is a standard queue based breadth-first traversal. On visiting an equivalence class $[\![w]\!]_O$ on level i, we assign a "traversal number" to this equivalence class. This traversal number is the smallest not yet used number on the i-th level. Since both searches are synchronized, in every step the same traversal number is assigned by each individual breadth-first search. So a periodic numbering is computed incrementally.

From the current equivalence class, the breadth-first search can use an outgoing or incoming transition to proceed. It halts when it cannot extend the traversal without leaving its level range or it encounters an equivalence class that has been labeled already. In the latter case, both PBFS have to encounter an equivalence class with the same number on level i and $i + k$ respectively.

It can happen that the PBFS fails because the choice of $[\![v]\!]_O$ as starting point on level $m + k$ is wrong. To find the right one, we simply repeat the procedure for every equivalence class $[\![v]\!]_O$ on level $m + k$ until we find the correct one.

In the case that the behavior graph consists of several connected components, we examine each connected component individually. As a result, we gain (partial) labelings, which can easily be combined to a complete periodic description. However, if one of the connected components could not be labeled,

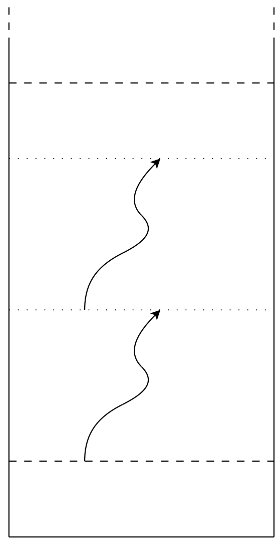


Figure 5.9: Schematic view of a parallel breadth-first search

we know that the current values of m and k fail to yield a periodic description. The connected components can be identified by using simple breadth-first searches starting from every equivalence class of level m.

If the level of the table is greater than $m+2k-1$, we still need to verify that the computed repeating numbering can be continued on the not yet considered levels. However, once a repeating structure is detected such a numbering can easily be obtained by copying the numbering from the levels m to $m+2k-1$.

This procedure computes a periodic description α of $\mathcal{A}_O$ if the offset m and the period k is given. Thus, it is left to show how the mentioned oracle can be simulated. We address this task by trial and error and try every reasonable value for m and k. Two nested loops can simulate the oracle: In the outer loop we try every offset $m = 0, \dots, t-2$ (we need at least the two upper levels for the PBFS) in ascending order. In the inner loop we try every possible period $k = 1, \dots, \lfloor \frac{t-m}{2} \rfloor$ (the period has to be small enough to allow a PBFS) also in ascending order. It is clear that a PBFS fails if the offset and period cannot be used to describe $\mathcal{A}_O$ periodically but it finds a periodic description if there is one. So it is possible to compute all periodic descriptions of $\mathcal{A}_O$. The whole procedure is done by the Function `computeDescriptions` shown on page 122.

After having computed all periodic descriptions α, we use them to construct the VCAs $\mathcal{A}_\alpha$. These VCAs are our conjectures. It is clear that, if $t \geq m+2k-1$, then one of these conjectures is the VCA $\mathcal{A}$ we are looking for. If we choose the descriptions in ascending order w.r.t. the offset as first and the period as second criterion, then the first equivalent conjecture found uses surely the characteristic description of L. This procedure guarantees that the VCA returned by the learner has threshold m. However, it can happen that $\mathcal{A}_O$ does not have any periodic description. In this case, we use the DFA $\mathcal{A}_O$ itself treated as a t-VCA as conjecture.[5]

[5]To treat the DFA $\mathcal{A}_O$ as a (partial) t-VCA, we simply use the same set of states, add a counter and define transitions δ_i from equivalence classes of the i-th level

```
Compute 𝒜_O;
Let Descr := computeDescriptions(𝒜_O);
Let ce = ∅;
if Descr ≠ ∅ then
    forall α ∈ Descr in ascending order do
        Let 𝒜 be the m-VCA 𝒜_α;
        if the MAT replies with a counter-example w to an equivalence
        query with 𝒜 then
            ce := ce ∪ {w};
        else
            Break;
        end
    end
else
    Let 𝒜 be the t-VCA gained from 𝒜_O;
    if the MAT replies with a counter-example w to an equivalence query
    with 𝒜 then
        ce := ce ∪ {w};
    end
end
if no equivalent VCA was found then
    Choose a counter-example w uniform randomly from ce;
    add(O, w);
end
```

Algorithm 9: The second step of the learner for VCA-acceptable languages

For each conjecture, we ask the MAT for equivalence. If one of these VCAs is equivalent, then we return it. If only counter-examples are returned, then the current level t is not high enough to find the characteristic description. In this case, we uniform randomly choose a counter-example and add it as well as all of its prefixes to the stratified observation table. We argue later that the level of the table is thereby increased. The adding is done by the Function `add` shown on page 130. This function also updates the table. The whole second step of our learner is shown in pseudo code as Algorithm 9.

Proof of correctness

We now turn on proving the main result of this section.

Theorem 5.21. *Let L be a VCA-acceptable language, whose characteristic description α has offset m and period k. The learner for VCA-acceptable languages eventually terminates and returns an m-VCA accepting L.*

To see that our learner for VCA-acceptable languages is correct, note that, if it ever terminates, then the returned VCA is clearly capable of recognizing L. This is ensured by the loop condition of the learner's main loop. Hence, we have to show that the learner terminates eventually.

We split the proof of Theorem 5.21 up into several parts. We first show that the learner eventually computes the congruence $\sim_L|_t$ for a fixed level t.

```
Input: A conjecture 𝒜_O representing BG_L|_t
Output: A set of periodic descriptions of 𝒜_O
Descr = ∅;
for m = 0, ..., t − 2 do
    for k = 1, ..., ⌊(t−m)/2⌋ do
        forall connected components C from level m to m + 2k − 1 of
        BG_L|_t do
            Choose an equivalence class ⟦u⟧_O of C on level m;
            forall equivalence classes ⟦v⟧_O of C on level m + k do
                Perform a PBFS starting with ⟦u⟧_O and ⟦v⟧_O;
                if the PBFS was successful then
                    Compute the (partial) labeling;
                    Break;
                end
            end
        end
        if the PBFS succeeded on all connected components then
            Compute a periodic description α from partial labelings;
            Descr := Descr ∪ {α};
        end
    end
end
return Descr;
```

Function `computeDescriptions(`$\mathcal{A}_O$`)`

Then, we show that in this case the conjecture $\mathcal{A}_O$ is isomorphic to $BG_L|_t$, thus, allowing us to use $\mathcal{A}_O$ to search for the characteristic description of L. It is clear that one of the periodic descriptions of $\mathcal{A}_O$ is the characteristic description of L if the level of the stratified observation table is at least $m+2k-1$. The processing of periodic descriptions in ascending order guarantees that the returned VCA has a threshold of exactly m. To see that the level eventually reaches the value $m+2k-1$, we finally show that the level of the stratified observation table is increased every time a counter-example is added in the second step.

Let us first prove that the learner eventually computes the congruence $\sim_L |_t$ for a fixed level t. From Observation 5.19 we know that a stratified observation table can only be not closed or not consistent at most $K \cdot t$ times. Thus, it is left to show that the processing of counter-examples to partial equivalence queries eventually results in a conjecture equivalent to $BG_L|_t$. This is done in the following lemma.

Lemma 5.22. *Let O be a stratified observation table up to a fixed level t. Then, there can be at most $K \cdot t$ wrong conjectures $\mathcal{A}_O$.*

Proof of Lemma 5.22. If the MAT is provided with a wrong conjecture, it returns two counter-examples uw, vw with $u \sim_O v$ and $uw \in L \Leftrightarrow vw \notin L$. Since $u, v \in \Sigma_{0,t}^*$, we know that the function `addPartial` adds both u and v to $R_{cv(u)}$ and w to $S_{cv(u)}$. Hence, the w-entry of the rows of u and v are distinct and

u and v are no longer O-equivalent. The number of O-equivalence classes and $index(\sim_O)$ increases.

Since $index(\sim_O) \leq index(\sim_L|_t) = K \cdot t$, this can happen at most $K \cdot t$ times. Moreover, from the fact $u \sim_L v \Rightarrow u \sim_O v$ we deduce that $u \sim_L v \Leftrightarrow u \sim_O v$ holds if $index(\sim_O) = index(\sim_L|_t)$. Thus, after at most $K \cdot t$ wrong conjectures the learner finds the correct one.

□

It is not hard to verify that a conjecture $\mathcal{A}_O$ is an isomorphic representation of $BG_L|_t$ if the MAT replies with "yes" to a partial equivalence query. Moreover, Observation 5.16 tells us that after reaching the level $m + k - 1$ the equivalence $\sim_L|_t$, and hence $\sim_O$, contains all L-equivalence classes up to this level. These properties allow us to use the conjecture $\mathcal{A}_O$ to search for the characteristic description of L.

Lemma 5.23. *Let O be a stratified observation table up to a fixed level t. If the MAT replies "yes" to a partial equivalence query with the conjecture $\mathcal{A}_O$, then $\mathcal{A}_O$ is isomorphic to $BG_L|_t$.*

Proof of Lemma 5.23. If the MAT replies with "yes" to a partial equivalence query with $\mathcal{A}_O = (Q_O, \Sigma, q_0^O, \delta_O, F_O)$, then

$$u \sim_O v \Leftrightarrow u \sim_L v$$

holds for all $u, v \in \Sigma_{0,t}^*$. Thus, the mapping

$$\varphi : \; Q_O \to Q_L|_t \;\; \text{with} \;\; \varphi([\![u]\!]_O) = [\![u]\!]_L$$

for all $[\![u]\!]_O \in Q_O$, where $Q_L|_t$ is the state set of $BG_L|_t$, is surely an isomorphism.

□

To see that the level of a stratified observation table is increased every time a counter-example is added, we show that each counter-example has a prefix with counter value greater than t. Intuitively, this has to hold since the fact that we learned $\sim_L|_t$ correctly ensures that the conjecture provided by the learner works correctly on all words in $\Sigma_{0,t}^*$. Thus, a counter-example has to force a conjecture to increment the counter value up to a value where we do not have knowledge about the unknown language yet. This intuition is summed up in the following lemma.

Lemma 5.24. *If the MAT replies a counter-example w to an equivalence query with the VCA $\mathcal{A}$, then there is a prefix $u \in pref(w)$ such that $cv(u) > t$.*

As mentioned, to proof Lemma 5.24 we first show formally that a conjecture works correctly on all words of $\Sigma_{0,t}^*$.

Lemma 5.25. *Let O be a stratified observation table with level t and $\mathcal{A}_O$ isomorphic to $BG_L|t$. Then, the following holds:*

(a) *Let α be a periodic description of $\mathcal{A}_O$ and $\mathcal{A} = \mathcal{A}_\alpha$ the VCA constructed by using Construction 5.13. Then,*

$$((\nu([\![\varepsilon]\!]_O), k-1), 0) \xrightarrow{w}_{\mathcal{A}} \begin{cases} ((\nu([\![w]\!]_O), k-1), cv(w)) \\ \text{, if } cv(w) < m \\ \\ ((\nu([\![w]\!]_O), (cv(w) - m) \mod k), cv(w)) \\ \text{, if } cv(w) \geq m \end{cases}$$

holds for all $w \in \Sigma^*_{0,t}$.

(b) *Let $\mathcal{A}$ be the DFA $\mathcal{A}_O$ treated as VCA. Then,*

$$([\![\varepsilon]\!]_O, 0) \xrightarrow{w}_{\mathcal{A}} ([\![w]\!]_O, cv(w))$$

holds for all $w \in \Sigma^*_{0,t}$.

Part (a) of Lemma 5.25 shows the close relation between the VCA $\mathcal{A}_\alpha$ constructed from an ultimately periodic description α of BG_L and $\mathcal{A}_{\alpha'}$ constructed from a periodic description α' of $BG_L|_t$. To be precise, the only difference between Lemma 5.25 part (a) and Lemma 5.14 is that part (a) of Lemma 5.25 only holds for words of $\Sigma^*_{0,t}$. This is because α describes the whole behavior graph while α' only describes the graph up to level t. Thus, we can only ensure that $\mathcal{A}_{\alpha'}$ works correctly on words of $\Sigma^*_{0,t}$. The behavior of $\mathcal{A}_{\alpha'}$ on words with counter value higher than t is perhaps wrong.

Proof of Lemma 5.25.

(a) The proof of part (a) is an exact copy of the proof of Lemma 5.14. The fact that $\mathcal{A}_O$ is isomorphic to $BG_L|_t$ and that α is a periodic description of $BG_L|_t$ ensures that the VCA $\mathcal{A}$ has all necessary information to simulate $BG_L|_t$ on words of $\Sigma^*_{0,t}$ correctly.

(b) The correctness of part (b) follows directly from the isomorphism between $\mathcal{A}_O$ and $BG_L|_t$. To treat $\mathcal{A}_O$ as a t-VCA, we simply adapt the transitions and add a counter but the structure of $\mathcal{A}_O$ is not changed. As long as the VCA does not exceed counter value t, it works in exactly the same way as BG_L. However, we skip the details of this straightforward proof.

□

Since we know that the conjectures work correctly on words of $\Sigma^*_{0,t}$, it is now easy to prove Lemma 5.24.

Proof of Lemma 5.24. First, recall that an equivalence query is not asked until the MAT returns “yes” to a partial equivalence query. Thus, we know that $\sim_O = \sim_L|_t$ holds.

Now, assume that $cv(u) \leq t$ for all prefixes $u \in pref(w)$. Since $w \in L(\mathcal{A}) \Leftrightarrow w \notin L$, we know that $cv(w) = 0$. We distinguish two cases depending on how the conjecture $\mathcal{A}$ is constructed.

(a) Assume that $\mathcal{A} = \mathcal{A}_\alpha$ for a periodic description α. The application of part (a) of Lemma 5.25 yields

$$((\nu([\![\varepsilon]\!]_O), k-1), 0) \xrightarrow{w}_{\mathcal{A}} ((\nu([\![w]\!]_O), k-1), 0)$$

since $w \in \Sigma_{0,t}^*$. Moreover, since $\sim_O = \sim_L |_t$ and from Construction 5.13 we know

$$(\nu(\llbracket w \rrbracket_O), k-1) \in F_{\mathcal{A}} \Leftrightarrow w \in L \ .$$

Hence, we gain $w \in L(\mathcal{A}) \Leftrightarrow w \in L$. This contradicts the fact that w is a counter-example.

(b) Now, assume that $\mathcal{A}$ is the DFA $\mathcal{A}_O$ treated as VCA. Then, we know

$$(\llbracket \varepsilon \rrbracket_O, 0) \xrightarrow{w}_{\mathcal{A}} (\llbracket w \rrbracket_O, 0)$$

from applying part (b) of Lemma 5.25. Since $\sim_O = \sim_L |_t$, we obtain

$$\llbracket w \rrbracket_O \in F_{\mathcal{A}} \Leftrightarrow w \in L$$

and, finally, $w \in L(\mathcal{A}) \Leftrightarrow w \in L$. Again, this contradicts the fact that w is a counter-example.

□

Hence, the level of a stratified observation table is increased by at least one every time a counter-example to an equivalence query is added to the table and t eventually reaches or exceeds the value $m + 2k - 1$. In fact, the increment of t is determined by the counter-examples returned by the MAT to an equivalence query. Thus, we are eventually able to find the characteristic description and can construct a VCA accepting L. This shows the termination and, hence, the correctness of our learner for VCA-acceptable languages.

Remark 5.26. The size of the resulting m-VCA for a VCA-acceptable language L, i.e. its number of states, is $m \cdot K$ where m is the threshold of the resulting VCA and K the slenderness index of BG_L.

◁

Complexity of the learner

Let us now discuss the complexity of the learner presented in this section. For a VCA-acceptable target language L the complexity depends on some parameters determined by characteristics of the language and the MAT.

The characteristics of L are the slenderness index K of its behavior graph as well as the offset m and the period k of the characteristic description of L. In contrast to the number of states or the threshold of a (minimal) VCA accepting L, we claim that that K, m and k are more natural parameters describing L. Particularly this holds since these parameter are unambiguous for each VCA-acceptable language.

We also have to take the answers of the MAT to both types of equivalence queries into account. Since each counter-example should be processed at least once completely, the length of counter-examples (or, more precisely, the length of the longest counter-example) has an influence on the runtime of any learner. Furthermore, the highest counter value of a counter-example determines the level of the learner's stratified observation table, and thus the total size of the table. The consideration of these parameters takes the "inefficiency" of the MAT into account.

Theorem 5.27. *Let L be a VCA-acceptable language, whose characteristic description α has offset m and period k, and let K be the slenderness index of the behavior graph BG_L. Moreover, let l be the length of the longest counter-example returned on a partial equivalence query and c the highest counter value of a counter-examples' prefix returned on an equivalence query. Then, an m-VCA $\mathcal{A}$ with $L(\mathcal{A}) = L$ can be learned in time polynomial in the parameters K, m, k and l, c. Moreover, the learner asks $\mathcal{O}(K^2 \cdot c^4 \cdot l^2)$ membership queries, $\mathcal{O}(K \cdot c^2)$ partial equivalence queries and $\mathcal{O}(c^3)$ equivalence queries.*

Proof of Theorem 5.27. From the proof of correctness we know that the learner eventually terminates and returns an m-VCA $\mathcal{A}$ accepting L. To prove the claimed complexity, let us first consider the number of main loops that are executed. Since the level of the table is increased only if a counter-example to an equivalence query is added, the maximal counter value of a prefix of such a counter-example determines the maximal level of the table. Thus, the number of main loops is bounded by c.

Within each loop, the table is extended because it is not closed or not consistent or a counter-example has to be processed. Let us have a detailed look at each operation.

- If O is not closed, then a new representative is added to R, thus, increasing R by one. From Observation 5.19 we know that this can happen only $K \cdot t$ times in each main loop. This yields an upper bound of $K \cdot c$ since $t \leq c$ always holds.
- If O is not consistent, then a new sample is added to S and, hence, increasing S by one. With the same argumentation as in the closed case this can only happen $K \cdot c$ times.
- If $\mathcal{A}_O$ is not equivalent to $BG_L|_t$, then two counter-examples u and v are added to O. Thereby, both R and S are increased by at most $2 \cdot l$ because every decomposition of u and v is possibly added to O and the lengths of u and v are bounded by l. From Lemma 5.22 we know that $\mathcal{A}_O$ can be at most $K \cdot c$ times nonequivalent to $BG_L|_t$.
- If no equivalent VCA is found, then a counter-example is added. Thus, both R and S are increased by at most l.

To sum up, both the number of representatives and the number of samples increases by

$$\begin{aligned}
\mathcal{O}(2 \cdot K \cdot c \cdot 1 + K \cdot c \cdot 2 \cdot l + l) &= \mathcal{O}(2 \cdot K \cdot c \cdot (1 + l) + l) \\
&= \mathcal{O}(K \cdot c \cdot l)
\end{aligned}$$

in each main loop. Since there are $\mathcal{O}(c)$ main loops, the size of the table is $\mathcal{O}([K \cdot c^2 \cdot l]^2) = \mathcal{O}(K^2 \cdot c^4 \cdot l^2)$. Thus, the learner asks at most $\mathcal{O}(K^2 \cdot c^4 \cdot l^2)$ membership queries. Moreover, the number of partial equivalence queries can be bounded by $\mathcal{O}(K \cdot c^2)$ since there are at most $K \cdot c$ partial equivalence queries in each main loop.

To estimate the number of equivalence queries, we have to consider the number of periodic descriptions that can be found in each loop. A closer look at the Function `computeDescritions` shows that the outer loop is executed t times while the inner loop is executed t times during each outer loop. Since a periodic description is perhaps discovered in an inner loop, there are $\mathcal{O}(t^2)$ periodic descriptions and, thus, equivalence queries in each learner's main loop. Hence, the number of equivalence queries is bounded by $\mathcal{O}(c^3)$.

To prove a polynomial runtime, we have to show that all tasks performed during a single learner's main loop have polynomial runtime. It is not hard to verify that the construction of $\mathcal{A}_O$ can be done in time polynomial in the size of the table. Moreover, the construction of a conjecture $\mathcal{A}_\alpha$ can also be done in time polynomial in the size of the table. This can happen at most c^2 times during each main loop.

To estimate the runtime of the Function `computeDescriptions`, let us first assume that the whole behavior graph is connected. Then, a PBFS is executed at most c^2 times. Within each execution, we may have to start with each of the K many equivalence classes on level $m+k$. A PBFS itself (and the following computation of a periodic description) may accesses all states of $BG_L|_t$ and, hence, has a runtime of at most $\mathcal{O}(K \cdot c)$. Thus, the Function `computeDescriptions` runs in time $\mathcal{O}(K^2 \cdot c^3)$. If the behavior graph consists of several, at most K, connected components, we have to repeat the search for a repeating labeling. However, each of these components contains fewer states and, thus, produces fewer costs in each individual PBFS. An amortized analysis shows that it makes no difference whether the graph is fully connected or it consists of several connected components. Overall, the Function `computeDescriptions` runs in time $\mathcal{O}(K^2 \cdot c^3)$.

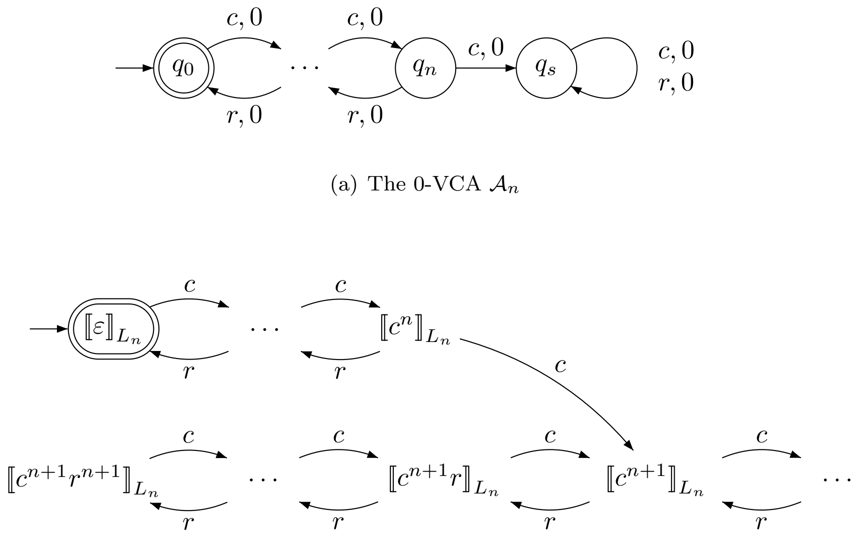
Overall, the runtime of each main loop is polynomial in K, l and c and, thus, the overall runtime of the learner is polynomial in K, l and c.

□

Note that the size of the table can be decreased dramatically by a factor in $\mathcal{O}(l^2)$ if we assume that the MAT returns u, v and w to a partial equivalence query instead of just uw and vw. In this case, we do not need to compute the correct decomposition of the counter-examples and, thus, do not need to add all decompositions to the table. In this case, it is sufficient to add u and v to $R_{cv(u)}$ and w to $S_{cv(u)}$.

Finally, let us consider the connection between the value c and the offset m and the period k of the characteristic description. We know that the level of a stratified observation table has to be at least $m + 2k - 1$ to find the periodic description necessary to construct an equivalent VCA. Thus, $m + 2k - 1 \leq c$ surely holds. Moreover, if we assume that the MAT provides "good" counter-examples, in this case that means counter-examples with "minimal" counter values of their prefixes, then $m + 2k - 1 = c$ holds. Thus, the runtime of the learners is also polynomial in the parameters m and k.

Even if the runtime of the learner for VCA-acceptable language is polynomial in the language's parameters, the learner may not result in a VCA with the smallest threshold possible. This is because a VCA can use its states to perform actions up to a fixed counter value, say n, which yields a characteristic description with offset at least n. In any case, a high number of states on the one hand and a high threshold on the other hand reflects a high "complexity" of



(a) The 0-VCA $\mathcal{A}_n$

(b) The behavior graph BG_{L_n}

Figure 5.10: A 0-VCA and the behavior graph for the languages $L_n = \Sigma^*_{0,n}$ for $n \geq 0$

the language. Formally, this phenomenon is stated in Lemma 5.28. The proof of this lemma follows our intuitive reasoning.

Lemma 5.28. *For each $n \geq 0$ there is a 0-VCA acceptable language L_n such that the learner for VCA-acceptable languages results in an n-VCA.*

Proof of Lemma 5.28. For each $n \geq 0$ consider the language $L_n = \Sigma^*_{0,n}$. The languages L_n are 0-VCA acceptable by using the 0-VCAs $\mathcal{A}_n$ depicted in Figure 5.10(a). The behavior graph of L_n is depicted in Figure 5.10(b). It is not hard to verify that the characteristic description of L has offset n and, thus, the learner returns an n-VCA.

$\square$

Lemma 5.28 also helps to develop a further understanding of the "complexity" of VCA-acceptable languages. It is not hard to see that the configuration graph $G_{\mathcal{A}_n}$ and the behavior graph BG_{L_n} are isomorphic. This observation supports our claim that the size of the behavior graph , i.e. the slenderness index K as well as the offset m and the period k of the characteristic description of L, is a reasonable measurement for the complexity of a VCA-acceptable language.

5.3.2 Application to the validation of parenthesis word representations

We apply the learner for m-VCA-acceptable languages to the setting of validating parenthesis word representations as follows. If provided with a MAT for a recognizable DTD d, the learner results in an m-VCA that accepts exactly the parenthesis word representations valid w.r.t. d. To gain an equivalent DFA, we first need to reduce the threshold to $m = 1$ and then apply the transformation presented Construction 5.3.

To reduce the threshold of an m-VCA, we use a method presented by Bárány, Löding and Serre [BLS06]. Under the assumption that there exists an equivalent m'-VCA for some $m' < m$, their construction yields such an m'-VCA. The basic idea is to guess the behavior of the m-VCA on counter values greater than m' and simulate the m-VCA nondeterministically. If the counter value drops below m', the guess can be verified. After this construction, the resulting VCA is determinized. The complexity of this method is doubly exponential in the number of states of the given m-VCA.

It is not hard to verify that Bárány's, Löding's and Serre's threshold reduction method yields an 1-VCA in the context of validation parenthesis word representations if the given m-VCA is equivalent to a recognizable DTD. In this case, Theorem 5.2 tells us that there exists an 1-VCA equivalent to d. We simply apply the threshold reduction method where we set $m' = 1$.

Remark 5.29. The complexity of the learner for VCA-acceptable languages and the subsequent threshold reduction determine the complexity of this approach. The transformation of the 1-VCA into a DFA can be done in linear time and can, therefore, be neglected in asymptotic considerations.

1. Referring to Theorem 5.27, the runtime of the learner for VCA-acceptable languages is polynomial in the characteristic parameters of the target language and, hence, in the "complexity" of the given DTD. In consideration of Remark 5.26 this does also hold for the size, i.e. the number of states, of the resulting m-VCA. However, a sound definition of "complexity of a DTD" and the connection between this complexity and the parameters of the language $L^{()}(d)$ is still an open question and requires further research.

2. As already mentioned, the complexity of the threshold reduction method is doubly exponential in the number of states of the given m-VCA.

All in all, the application of our learning method to the validation of parenthesis word representations has a runtime that is doubly exponential in the complexity of the given DTD. However, we expect our method to perform much better in practice than a trial and error approach.

$\triangleleft$

```
Input: A stratified observation table O
Input: Two words u = a_1 ... a_n, v = b_1 ... b_m

forall i = 1, ..., n do
    Let u' = a_1 ... a_i and u'' = a_{i+1} ... a_n;
    if cv(u') ≤ t then
        R_i = R_i ∪ {u'};
        S_i = S_i ∪ {u''};
        Update T by asking membership queries for all new entries;
    else
        Break;
    end
end
forall i = 1, ..., m do
    Let v' = b_1 ... b_i and v'' = b_{i+1} ... b_m;
    if cv(v') ≤ t then
        R_i = R_i ∪ {v'};
        S_i = S_i ∪ {v''};
        Update T by asking membership queries for all new entries;
    else
        Break;
    end
end
```

Function `addPartial`(O, u, v)

```
Input: A stratified observation table O
Input: A word u = a_1 ... a_n

forall i = 1, ..., n do
    Let u' = a_1 ... a_i and w = a_{i+1} ... a_n;
    if i > t then
        t := t + 1;
        R_t = S_t = ∅;
    end
    R_i = R_i ∪ {u'};
    S_i = S_i ∪ {w};
    Update T by asking membership queries for all new entries;
end
```

Function `Add`(O, u)

6 Conclusion

Inspired by the work of Segoufin and Vianu on the validation of streaming XML documents against DTDs under a constant memory constraint, we addressed the task of validating serializations of trees by using finite automata. As serializations, we considered XML as well as parenthesis word representations. Since no direct construction method of a DFA from a recognizable DTD is known, our approach was the use of learning algorithms to obtain such DFAs.

We developed learning algorithms not for the special case of validating tree serializations but considered two general language classes: The class of regular languages with "don't cares" and the class of VCA-acceptable languages. For the class of regular languages with "don't cares" we developed several heuristic algorithms, which we applied to learn DFAs that validate XML word representations. For the class of VCA-acceptable languages we presented a learning algorithm, for which we proved its correctness and a polynomial runtime. This algorithm was applied to the task of learning DFAs that validates parenthesis word representations. By means of the learning algorithm for VCA-acceptable languages, we succeeded in providing an efficient construction method for DFAs that validate parenthesis word representations against DTDs. Additionally, the learners for regular languages with "don't cares" provide a construction method for DFAs that validate XML word representations but a formal proof of their correctness was not addressed in this thesis.

Besides the theoretic work on learning algorithms, this thesis also covers a practical part. In this, we implemented both our heuristics for learning regular languages with "don't cares" and our MAT for XML word representations. The intention of this implementation was to provide a proof-of-concept for our method of constructing DFAs that validate XML word representations. Experiments on the RWTH Aachen's HPC revealed that for small DTDs DFAs could be learned but our heuristics performed worse on more complex DTDs.

Further research

Since an efficient construction method for DFAs that validate XML word representations against recognizable DTDs is still an open question, an approach similar to our method for parenthesis word representations seems possible: Instead of learning a DFA directly, one tries to learn a *restricted visibly pushdown automaton*, which is subsequently transformed into a DFA. Such restricted visibly pushdown automata are a subclass of pushdown automata where the input symbol does not only determine the kind of the performed stack operation but also determines which symbol has to be pushed and popped. Intuitively, a restricted visibly pushdown automaton has to push the current input symbol on its stack on reading a call symbol. Moreover, the reading of a return symbol is only allowed if the current input symbol matches the symbol on top of the stack. It is not hard to verify that, in analogy to Theorem 5.2, there exists a DFA recognizing a given DTD d if and only if there exists a restricted visibly pushdown automaton recognizing d. Furthermore, a restricted visibly pushdown automaton recognizing d can efficiently be transformed into a DFA recognizing d and vice versa. Again, the restricted visibly pushdown automaton uses its stack only to check whether the input is an XML word representations but performs the validation in a "regular way".

The connection to the validation of parenthesis word representations becomes immediately clear if one thinks of the tags of a DTD as opening and closing parenthesis: Parenthesis word representations have only one type of parenthesis and can be validated by a visibly one-counter automaton, which uses only one type of stack symbol. In contrast, XML word representations can be validated by a restricted visibly pushdown automaton, which uses as many stack symbols as there are tags (under the additional constraint that the symbol pushed or popped matches the current input symbol).

Considering this, the next step is the development of a learning algorithm for the mentioned restricted class of visibly pushdown automata. Moreover, a precise characterization of recognizability for both the XML and the parenthesis word representation based on (structural) properties of a DTD is still an open question.

Bibliography

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM Press.

[AMN05] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[BDG97] José L. Balcázar, Josep Díaz, and Ricard Gavaldà. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, (21):592–597, June 1972.

[BLS06] Vince Bárány, Christof Löding, and Olivier Serre. Regularity problems for visibly pushdown languages. In *STACS*, pages 420–431, 2006.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (fourth edition). Technical report, W3C, 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[BR87] Piotr Berman and Robert Roos. Learning one-counter languages in polynomial time (extended abstract). In *FOCS*, pages 61–67, 1987.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247, 2000.

[FB93] Amr F. Fahmy and Alan W. Biermann. Synthesis of real time acceptors. *Journal of Symbolic Computation*, 15(5-6):807–842, 1993.

[FR95] A. F. Fahmy and R. Roos. Efficient learning of real time one-counter automata. In K. P. Jantke, T. Shinohara, and Th. Zeugmann, editors, *Proceedings of the 6th International Workshop on Algorithmic Learning Theory ALT'95*, volume 997, pages 25–40, Berlin, 18–20 1995. Springer.

[GL] Olga Grinchtein and Martin Leucker. Learning finite-state machines from inexpirienced teachers.

[GO92] P. Gracía and J. Oncina. Inferring regular languages in polynomial update time. In N. Pérez de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. ACM Press, 2nd edition, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, August 1994.

[Le 06] Daniel Le Berre. Minilearningheapexpsimp / sat4j. Technical report, Université d'Artois, 2006. http://fmv.jku.at/sat-race-2006/descriptions/1-sat4j.pdf.

[Lee96] Lillian Lee. Learning of context-free languages: A survey of the literature. Technical Report TR-12-96, Harvard University, 1996. Available via ftp, ftp://deas-ftp.harvard.edu/techreports/tr-12-96.ps.gz.

[Pap95] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1995.

[RS89] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420, New York, NY, USA, 1989. ACM Press.

[SS07] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007.

[SV02] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–64, New York, NY, USA, 2002. ACM Press.

Index