FOUNDATIONS OF ACTIVE AUTOMATA LEARNING: AN ALGORITHMIC PERSPECTIVE

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

MALTE ISBERNER

Dortmund

2015

Tag der mündlichen Prüfung: 29.09.2015
Dekan: Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Frits Vaandrager

Foundations of Active Automata Learning

An Algorithmic Perspective

Malte Isberner

Tuesday 6th October, 2015

Abstract

The wealth of model-based techniques in software engineering—such as model checking or model-based testing—is starkly contrasted with a frequent lack of formal models in practical settings. Sophisticated static analysis techniques for obtaining models from a source- or bytecode representation have matured to close this gap to a large extent, yet they might fall short on more complex systems: be it that no sufficiently robust decision procedures are available, or that the system performs calls to external, closed source libraries or even remote web services.

Active automata learning has been proposed as a means of overcoming this problem: by executing test cases on a system, finite-state machine models reflecting a portion of the *actual* runtime behavior of the targeted system can be inferred. This positions active automata learning as an *enabler technology*, extending the range of application for a whole array of formal, model-based techniques. Its usefulness has been proven in many different subfields of formal methods, such as black-box model checking, test-case generation, interface synthesis, or compositional verification. In a much-noted case study, active automata learning played a key role in analyzing the internal structure of a botnet with the aim of devising countermeasures.

One of the major obstacles of applying active automata learning in practice is, however, the fact that it is a rather *costly* technique: to gain sufficient information for inferring a model, a large number of test cases need to be executed, which is also referred to as "posing queries." These test cases may be rather heavy-weighted, comprising high-latency operations such as interactions with hardware or remote network services, and learning systems of moderate size may take hours or days even when using algorithms with polynomial query complexities.

The costliness of the technique calls for highly efficient algorithms that do not waste any information. The reality is surprisingly different from that ideal: many active automata learning algorithms that are being used in practice—including the well-known L* algorithm, which was the first one with a polynomial query complexity—frequently resort to heuristics to ensure certain properties, resulting in an increased overall query complexity. However, it has rarely been investigated why or even if these properties are necessary to ensure correctness, or what violating them entails. Related is the observation that descriptions of active automata learning algorithms are often less-than-formal, and merely focus on *somehow* arriving at a correctness proof instead of motivating and justifying the single steps.

It is one of the stated goals of this thesis to change this situation, by giving a rigorously formal description of an approach to active automata learning that is independent of specific data structures or algorithmic realizations. This formal description allows the identification of a number of properties, some of which are necessary, while others are merely desirable. The connection between these properties, as well as possible reasons for their violation, are investigated. This leads to the observation that, while for each property there is an existing algorithm maintaining it, no algorithm manages to simultaneously maintain all desirable properties.

Based on these observations, and exploiting further insights attained through the formalization, a novel active automata learning algorithm, called TTT, is developed. The distinguishing

characteristic of TTT is that it eventually ensures that all desirable properties are maintained. This is realized based on a careful observation of how certain syntactic and semantic properties are related to each other, and how their violations can be exploited for further refinements.

The approach of developing an algorithm strictly adhering to principles identified as desirable in a formal framework yields a number of benefits: a proof that the TTT algorithm is the first *space-optimal* active automata learning algorithm is given, meaning there can be no algorithm with an asymptotically lower space complexity correctly accomplishing the same task. Since TTT maintains all observations (i.e., responses to queries) made throughout the learning process in its data structures, this theoretical result indicates a very economic handling of information, indicating that the algorithm indeed poses only those queries which are necessary. On the practical side, our evaluations show that TTT is superior to virtually every other learning algorithm. This especially applies if counterexamples are non-minimal (a situation frequently encountered in practice), and if furthermore not only the number of queries, but also their combined length is considered.

A further limitation of active automata learning is that it is restricted to regular languages (or systems whose behavior can be described by a regular language), at least in its classical formulation. Extensions have been proposed recently, mainly concerning the handling of data. In this thesis, we will investigate another dimension, namely context-free control structure: by presenting an algorithm for inferring *visibly pushdown automata*, we extend the applicability of active automata learning to systems with (recursive) calls and returns. In doing so, we further highlight the benefits of a rigorous formalization: identifying key similarities between regular and visibly pushdown languages provides what can be described as a clear recipe to build an algorithm for learning visibly pushdown languages, which furthermore allows leveraging many of the optimizations developed for the setting of regular languages.

We will thus not only describe a "simply working" algorithm for inferring visibly pushdown automata, but one that can be regarded as a visibly pushdown version of the TTT algorithm, called TTT-VPA. This algorithm has a similar space complexity and, according to a preliminary experimental evaluation, exhibits a similarly superior performance, especially in the presence of long counterexamples. While there is no wide range of other algorithms against which we can compare the performance of TTT-VPA, we evaluate the impact of those steps which can be regarded as characteristic for TTT, and show that they result in a significant performance increase also in the setting of visibly pushdown languages. This can be regarded as a clear indication that adhering to formally identified principles indeed pays off, and is the key to developing algorithms of superior practical performance.

Acknowledgements

First and foremost, I would like to thank Bernhard Steffen for his support and guidance over the past eight years. Thank you for introducing me to the beautiful field of active automata learning, tirelessly motivating, supporting and challenging me, and for making sure that I keep a balance between the formal and the intuitive.

I would like to thank Frits Vaandrager, for agreeing to act as my second referee on rather short notice and in spite of a very tight schedule, and for providing many helpful remarks which have been incorporated into the final version of this thesis.

My PhD studies would have been a lot less exciting without three fantastic summer internships in 2012, 2013, and 2014, which really had a great impact on my further development, both professionally and personally. I owe special thanks to Bengt Jonsson, Dimitra Giannakopoulou, and Vishwanath Raman for making them possible.

I would furthermore like to thank all my colleagues, in particular those from Dortmund, for creating a warm atmosphere at work, and for many fun conversations that brightened up my day-to-day life. Special thanks go out to Falk Howar, for many fruitful and stimulating discussions about automata learning and other things.

I am especially grateful to my family for their support and encouragement for as long as I can remember. Many thanks go out to Lisa Steinmann and Rebecca Doherty, for their effort of proofreading this thesis. Last but not least, I want to thank my girlfriend Maren Geske for her support, for tolerating me even when being completely caught up in my research, and for simply everything else.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CE	counterexample
DFA	deterministic finite automaton
DPDA	deterministic pushdown automaton
DT	discrimination tree
EQ	equivalence query
FSA	finite-state acceptor
FSM	finite-state machine
KV	Kearns and Vazirani's algorithm
LCA	lowest common ancestor
MAT	minimally adequate teacher
MQ	membership query
NFA	non-deterministic finite automaton
OP	Observation Pack
OT	observation table
PDA	pushdown automaton
RS	Rivest and Schapire's algorithm
RSFA	residual finite-state automaton
SEVPA	single-entry visibly pushdown automaton
VPA	visibly pushdown automaton
VPL	visibly pushdown language

Notation

General

Symbol	Meaning	See ...
$\mathbb{B}$	Set of Boolean values, $\mathbb{B} = \{0,1\}$	p. 7
$\mathbb{N}$	Set of non-negative integers, $\mathbb{N} = \{0,1,2,\ldots\}$	p. 7
$\mathbb{Z}$	Set of all integers, $\mathbb{Z} = \{0,1,-1,2,-2,\ldots\}$	p. 7
$\lvert X \rvert$	Cardinality of set X	p. 7
2^X	Power set of set X, $2^X = \{Y \mid Y \subseteq X\}$	p. 7
$f : X \rightharpoonup Y$	Partial function from X to Y	p. 8
$\operatorname{dom} f$	Domain of a (partial function) f	p. 8
$[x]_{\approx}$	Equivalence class of x wrt. equivalence relation $\approx$	p. 8
$\operatorname{ind}(\approx)$	Index of an equivalence relation $\approx$	p. 8
$X/{\approx}$	Quotient of X wrt. equivalence relation $\approx$	p. 8
$\sim_P$	Equivalence relation induced by partiton P	p. 8
$\sim_f$	Equivalence kernel of a function f	p. 8
$[x]_f$	Equivalence class of x wrt. $\sim_f$	p. 8

Words, Languages, Automata

Symbol	Meaning	See ...
Σ	Input alphabet	p. 9
a	Single input symbol, $a \in \Sigma$	p. 9
Σ^*	Set of words over alphabet Σ	p. 9
w	Single word, $w \in \Sigma^*$	p. 9
$\lvert w \rvert$	Length of a word $w \in \Sigma^*$	p. 9
ε	The empty word, i.e., the unique word of length 0	p. 9
Σ^+	Set of non-empty words over Σ, $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$	p. 9
$u \cdot v$, uv	Concatenation of u and v	p. 9
$U \cdot V$, UV	Concatenation lifted to sets $U, V \subseteq \Sigma^*$	p. 9
$\sqsubseteq_{pref}$ ($\sqsubset_{pref}$)	"is-(strict-)prefix-of" relation	p. 10
$\operatorname{Pref}(w)$	Set of all prefixes of w	p. 10
$\sqsubseteq_{suff}$ ($\sqsubset_{suff}$)	"is-(strict-)suffix-of" relation	p. 10

Symbol	Meaning	See ...
$\mathrm{Suff}(w)$	Set of all suffixes of w	p. 10
$\mathcal{A}$	NFA, DFA, or generic finite-state machine	pp. 12ff.
$\mathcal{D}$	Output domain, usually $\mathcal{D}=\mathbb{B}$ or $\mathcal{D}=\Omega^*$	p. 15
$Q_\mathcal{A}$	Set of states of automaton $\mathcal{A}$	pp. 12ff.
q	Single state, $q \in Q_\mathcal{A}$	pp. 12ff.
$q_{0,\mathcal{A}}$	Initial state of automaton $\mathcal{A}$, $q_{0,\mathcal{A}} \in Q_\mathcal{A}$	pp. 12ff.
$\Delta_\mathcal{A}$	Transition relation of NFA $\mathcal{A}$, $\Delta_\mathcal{A} \subseteq Q_\mathcal{A} \times \Sigma \times Q_\mathcal{A}$	p. 12
$\delta_\mathcal{A}$	(Extended) transition function of automaton $\mathcal{A}$	pp. 13ff.
$F_\mathcal{A}$	Final states of FSA $\mathcal{A}$, $F_\mathcal{A} \subseteq Q_\mathcal{A}$	pp. 12, 14
$\mathcal{M}$	Mealy machine	p. 15
Ω	Output alphabet	p. 15
$\gamma_\mathcal{M}$	Transition output function of Mealy machine $\mathcal{M}$	p. 15
$\lambda_\mathcal{A}$	Output function of automaton $\mathcal{A}$	p. 12
$\lambda_\mathcal{A}^q$	State output function of state q in automaton $\mathcal{A}$	p. 15
$\mathcal{A}[w]$	State in automaton $\mathcal{A}$ reached by word $w \in \Sigma^*$	p. 17
$\equiv, \equiv_\mathcal{A}$	Equivalence between states of automaton $\mathcal{A}$	p. 17
$\mathcal{L}(\mathcal{A})$	Language accepted by a DFA $\mathcal{A}$	p. 14
$\mathscr{L}_k$	Class of Chomsky type-k languages	p. 21

Active Automata Learning

General

Symbol	Meaning	See ...
λ	Target output function (over alphabet Σ)	p. 25
$\mathcal{A}$	(Canonical) target DFA, $\lambda = \lambda_\mathcal{A}$	p. 25
n	Number of states of $\mathcal{A}$, $n = \Sigma^*/\cong_\lambda$	p. 26
k	Size of the input alphabet, $k = \lvert\Sigma\rvert$	p. 26
m	Length of longest counterexample	p. 26
$\cong_\lambda$	Nerode congruence wrt. suffix output function λ	p. 23
$\mathcal{H}$	Inferred hypothesis	p. 25

Black-Box Abstractions

Symbol	Meaning	See ...
κ	Black-box classifier, $\kappa : \Sigma^* \to \mathscr{C}$	p. 28
$Ch_\kappa(u)$	Characterizing set (wrt. κ) of $u \in \Sigma^*$	p. 28
$Seps_\kappa(u, u')$	Separator set (wrt. κ) of $u, u' \in \Sigma^*$	p. 28

Symbol	Meaning	See ...
$sep_\kappa(u,u')$	Unique element in $Seps_\kappa(u,u')$ (if applicable)	p. 61
K_λ	Set of all valid black-box classifiers for output function λ	p. 28
$\mathcal{U}$	Short (or representative) prefixes, $\mathcal{U} \subset \Sigma^*$	p. 29
$\mathcal{R}$	Black-box abstraction, $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$	p. 29
$\mathscr{C}(\mathcal{R})$	Classes of black-box abstraction $\mathcal{R}$	p. 29
$\mathcal{V}$	Global set of distinguishing suffixes, $\mathcal{V} \subset \Sigma^*$	p. 29
$\sqsubseteq$ ($\sqsubset$)	(Strict) refinement relation between black-box classifiers and abstractions	p. 34
$\rho_\mathcal{R}(C)$	Representatives of a class $C \in \mathscr{C}(\mathcal{R})$, $\rho_\mathcal{R}(C) = C \cap \mathcal{U}$	p. 30
α	Abstract counterexample	p. 38
$\mathcal{E}$	Effect domain of an abstract counterexample α	p. 38
$\triangleright$	Effect relation of an abstract counterexample α	p. 38
l	Length of an abstract counterexample α	p. 38
η	Effect mapping of an abstract counterexample α, $\eta\colon \{0,\ldots,l\} \to \mathcal{E}$	p. 38
$\lfloor q \rfloor$	Access sequence (unique representative) of state $q \in Q_\mathcal{H}$	p. 43
$\lfloor w \rfloor_\mathcal{H}$	Access sequence of state reached by $w \in \Sigma^*$, $\lfloor w \rfloor_\mathcal{H} = \lfloor \mathcal{H}[w] \rfloor$	p. 44

Discrimination Trees

Symbol	Meaning	See ...
$\mathcal{T}$	Discrimination tree	p. 62
$\mathcal{N}_\mathcal{T}, \mathcal{I}_\mathcal{T}, \mathcal{L}_\mathcal{T}$	Nodes, inner nodes, leaves of discrimination tree $\mathcal{T}$	p. 62
$r_\mathcal{T}$	Root node of discrimination tree $\mathcal{T}$, $r_\mathcal{T} \in \mathcal{N}_\mathcal{T}$	p. 62
n	A node in a discrimination tree	pp. 62ff.
l	A leaf in a discrimination tree	pp. 62ff.
$Sig_\mathcal{T}(n)$	Signature of a node $n \in \mathcal{N}_\mathcal{T}$	p. 62
$Ch_\mathcal{T}(n)$	Characterizing set of a node $n \in \mathcal{N}_\mathcal{T}$	p. 62
$lca_\mathcal{T}(a,b)$	Lowest common ancestor of nodes a and b in $\mathcal{T}$	p. 64
$sep_\mathcal{T}(a,b)$	Separating discriminator of nodes a and b in $\mathcal{T}$	p. 64
$\pi(\mathcal{T})$	Block partition induced by discrimination tree $\mathcal{T}$	p. 65
B	A block $B \in \pi(\mathcal{T})$	p. 68
$depth(\mathcal{T})$	Depth of discrimination tree $\mathcal{T}$	p. 66

Visibly Pushdown Automata

Symbol	Meaning	See ...
$\widehat{\Sigma}$	Visibly pushdown alphabet, $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$	p. 119
Σ_{call}	Set of call actions	p. 119
c	Single call action, $c \in \Sigma_{call}$	p. 121ff.
Σ_{ret}	Set of return actions	p. 119
r	Single return action, $r \in \Sigma_{ret}$	p. 121ff.
Σ_{int}	Set of internal actions	p. 119
i	Single internal action, $i \in \Sigma$	p. 121ff.
β	Call-return balance, $\beta : \widehat{\Sigma}^* \to \mathbb{Z}$	p. 120
$\mathrm{MC}(\widehat{\Sigma})$	Set of call-matched words over $\widehat{\Sigma}$	p. 120
$\mathrm{MR}(\widehat{\Sigma})$	Set of return-matched words over $\widehat{\Sigma}$	p. 120
$\mathrm{WM}(\widehat{\Sigma})$	Set of well-matched words over $\widehat{\Sigma}$, $\mathrm{WM}(\widehat{\Sigma}) = \mathrm{MC}(\widehat{\Sigma}) \cap \mathrm{MR}(\widehat{\Sigma})$	p. 120
$\mathcal{L}_{\mathrm{WM}}(\mathcal{A})$	Well-matched language of VPA $\mathcal{A}$	p. 123
$L_{\mathcal{A}}$	Set of locations of VPA $\mathcal{A}$	p. 121
ℓ	Single location, $\ell \in L_{\mathcal{A}}$	p. 121ff.
$\ell_{0,\mathcal{A}}$	Initial location of VPA $\mathcal{A}$, $\ell_{0,\mathcal{A}} \in L_{\mathcal{A}}$	p. 121
Γ, $\Gamma_{\mathcal{A}}$	Stack alphabet (of VPA $\mathcal{A}$)	p. 121
γ	Single stack symbol, $\gamma \in \Gamma$	p. 121
σ	Stack contents, $\sigma \in \Gamma^*$	p. 121
$\delta_{call,\mathcal{A}}$	Call transition function of VPA $\mathcal{A}$	p. 121
$\delta_{ret,\mathcal{A}}$	Return transition function of VPA $\mathcal{A}$	p. 121
$\delta_{int,\mathcal{A}}$	Internal transition fucntion of VPA $\mathcal{A}$	p. 121
$\mathrm{CP}(\widehat{\Sigma})$	Set of context pairs over $\widehat{\Sigma}$	p. 125
$\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$	Set of $\mathcal{U}$-context pairs over $\widehat{\Sigma}$	p. 132

1. Introduction

Nearly thirty years ago, Dana Angluin published her seminal work *Learning Regular Sets from Queries and Counterexamples* [19], in which she proved that the class of regular languages could be learned efficiently (i.e., in time polynomial in the size of the canonical DFA for this language) using so-called *membership* and *equivalence queries*. More precisely, for an unknown regular language L, a *learner* can infer a model of the canonical DFA for L by asking polynomially many questions of the form "Is the word w in L?" and "Is L the language recognized by my current hypothesis DFA $\mathcal{H}$?" The problem solved by Angluin [19] is also referred to as *active automata learning* (sometimes also called *regular inference*). It is part of the field of *grammatical inference* [61] (or *grammar induction*), which is concerned with *learning* formal representations (i.e., automata or grammars) of *languages* in an abstract sense.

The positive learnability result for a complete and practically relevant class of languages received a lot of attention; at the time of writing, Google Scholar lists 1,500 citations for the above article. However, applications of the technique remained rare for a long time. A bibliographical survey by de la Higuera [60] lists map learning (i.e., an entity such as a robot inferring a map of its environment, as sketched by Rivest and Schapire [155]) as the only application of active automata learning as described above. The requirement of a teacher who must provide a definite and truthful answer to a query (i.e., noise cannot be tolerated) led most practical applications to focus on *passive* inference techniques instead [75, 168], where the teacher is replaced with a *sample set* containing labeled data, which the learner may access.

With the dawn of the new millennium came what can be described as a *renaissance* of active automata learning: the seminal works of Peled *et al.* on *black-box checking* [81, 149, 150], and by Steffen *et al.* on *test-based model generation* [84, 85, 101, 102], established a connection between active automata learning and the area of *formal methods*. By using active automata learning to generate models to be used by two widely-used formal, model-based techniques—model checking [24, 56] and model-based testing [39]—, the works paved the way for overcoming a frequently encountered, major obstacle of these techniques: the unavailability of such models in many scenarios. These initial works sparked a series of further investigations of the applicability of active automata learning in the context of formal methods, e.g., for interface synthesis [16, 71, 99], typestate analysis [173], or compositional verification [57, 70].[1]

The "adoption" of active automata learning by the formal methods community can by all means be described as fruitful: the plethora of practical applications inspired elaborate engineering efforts, greatly enhancing the efficiency in practical scenarios [48, 103, 130]. Challenges arising due to the characteristics of real-life software systems furthermore spawned research pushing the boundaries of the technique, resulting in algorithms for richer classes of models, e.g., adequately addressing phenomena such as time [79, 80] or data [1, 5, 45, 111]. Further evidence of the importance of formal methods for furthering the development of active automata learning, which however is of more anecdotal nature, is the fact that the 2010 ZULU competi-

[1] A more comprehensive overview can be found in Section 7.2.3.

tion [58], organized by members of the grammatical inference community, was actually won by a member of the formal methods community [94].

Despite these impressive improvements concerning the practicality and range of active automata learning, advancements on the purely algorithmic side remain rare. Even comparably recent works applying active automata learning in practice, the authors of which often put considerable engineering effort into speeding up the learning process (e.g., through parallelization [48, 96] or by exploiting domain-specific knowledge [27, 103, 130]), are oftentimes based on the original L* algorithm as described by Angluin [19]. This seems somewhat surprising, as algorithms with considerably better worst-case bounds (and from the experience of the author, also much better practical performance) have subsequently been proposed [115, 155]. Possible reasons for the relatively poor adoption of such improvements are that they are significantly harder to not only *implement* (in contrast to the rather simple L* algorithm), but also to *understand*.

Balcázar *et al.* [25], in their 1997 survey on active automata learning algorithms, pointed out that most of the original works on active automata learning hardly provide easy-to-grasp intuitions, and that "what makes the proof work" is often less than obvious. This is reflected in the fact that many active automata learning algorithms show a strongly heuristical nature: they resort to strategies that *somehow* work, in the sense that they guarantee progress or correctness, without however adequately addressing or even identifying the phenomena at hand. Poorer practical or worst-case performance is one of the consequences; more objectionable from a philosophical standpoint is that these heuristics actually miss what should be one of the central research question in active learning: *which are the questions that I* need to *ask?*

1.1. Research Questions

It has been pointed out above that a likely reason for many active automata learning algorithms resorting to heuristics is the lack of a precise understanding or even identification of the phenomena at hand, in particular when it comes to the analysis of counterexamples. This presumption is supported by the observation that the extent to which descriptions of active automata learning are truly formal is rather limited. However, only a strict mathematical characterization establishes a precise enough language which allows to reason about these phenomena in the first place. This gives rise to the first research question addressed in this thesis:

> *How can the phenomena encountered in active automata learning be characterized formally and independently of a concrete algorithmic realization, what is their significance, and what are desirable properties and characteristics that a learning algorithm should possess?*

Chapter 3 is dedicated to this question. A central insight that results from this consideration is that counterexamples are a manifestation of the more general concept of (reachability and output) inconsistencies. This challenges the typical approach of using observed inconsistencies to derive counterexamples, and suggests to regard *counterexample analysis* as merely a special case of *inconsistency analysis*.

While a precise mathematical formalization is prerequisite for devising efficient solutions, designing an algorithm involves much more, such as organizing and maintaining data efficiently. The second research question thus involves more than a straightforward application and implementation of the identified abstract concepts:

> *How can the insights gained through a rigorous formalization be translated into an efficient active learning algorithm, and how does the practical performance of an algorithm designed along these guidelines differ from existing algorithms?*

Several chapters in this thesis are related to this research question, as an *algorithm* cannot possibly be separated from the *data structures* it uses: the efficiency of most well-known algorithms is due to their cleverly exploiting the characteristics of specific data structures, and, conversely, the fact that it allows to efficiently solve certain problems is what constitutes the value of a data structure. Consequently, we will thus first study the data structure that enables efficient active automata learning algorithms in detail (Chapter 4), before describing how an active learning algorithm can be built on top of it in Chapter 5.

In its application in the context of formal methods, classical active automata learning often reaches its limits due to its restriction to finite-state systems. Several recent works investigate the possibility to extend it to certain classes of infinite-state systems. This gives rise to our third and final research question:

> *To what extent—and if so, how—can the mathematical formalization and the identified principles of efficient algorithm design be transferred to the active inference of richer classes of models, e.g., modeling infinite-state systems?*

This question will be addressed in Chapter 6, choosing *visibly pushdown automata* as a modeling formalism for infinite-state systems with recursion. As this question focuses on a *transfer* of concepts, an "incremental" consideration is justified, i.e., looking at what the (minimum) changes required for accommodating to the modified setting are, instead of building an independent, full-fledged theoretical framework from scratch.

1.2. Scope of This Thesis

This thesis is on a middle ground between theory and practice of automata learning: on one hand, our considerations are purely theoretical in that our assumptions do not go beyond those established by Angluin [19] for the so-called MAT framework. In particular, we do not concern ourselves with practical realizability of queries (including equivalence queries); this is the subject of several survey papers [167], tutorials [100], and some recent PhD theses [1, 93, 105, 138]. Understanding the connection between active automata learning on one hand and model-based techniques on the other hand is certainly helpful, in particular as a motivation, but not necessary to understand the technical content of this thesis. In terms of the contents, our requirement is a strictly mathematical characterization of the phenomena at hand, and, when it comes to algorithmic realization, a precise explanation of why every single step and query is necessary.

On the other hand, the *motivation* for the research presented in this thesis clearly originates from the practical applications of active automata learning in the context of formal methods. This is reflected, for instance, in the cost model that we apply for the worst-case analyses of algorithms: for a long time, it was common to consider the asymptotic number of membership queries required by a learning algorithm only (*query complexity*), as every query results in a single-bit answer (true/1 or false/0). From a practical perspective, however, it is clear that the

time required for realizing a query asymptotically grows at least linearly in its length. This motivates a worst-case analysis of the total number of symbols in all queries (*symbol complexity*), that we will present for all algorithms (cf. also Table A.1 in Appendix A). The motivation of generating (state-machine) models to be used with model-based techniques also justifies a narrow focus on only such techniques, and excluding other types of active learning (e.g., of Boolean formulae [22, 40] or Support Vector Machines [160]). Furthermore, this perspective motivates the considered extensions beyond DFAs and regular languages, namely Mealy machines, commonly used for modeling reactive systems [128], and visibly pushdown automata [11], which have been proposed as a model for programs with recursion.

1.3. Overview of the Contributions

Guided by the research questions listed above, the contributions presented in this thesis are the following:

- **Formalization of active automata learning:** A rigorous formalization of refinement-based active DFA learning is established. The mathematical precision of the presentation allows to naturally identify previously neglected phenomena, especially concerning the analysis of counterexamples. In particular, it is shown that counterexamples are manifestations of special cases of inconsistencies, which can be analyzed using dedicated techniques. The established mathematical framework furthermore provides guidelines for efficient algorithm design, while allowing to reduce proofs of the correctness and complexity of learning algorithms to casting them as instantiations of the framework.
- **Algorithmic advancements of classical active automata learning:** A novel, highly efficient active automata learning algorithm is presented, exploiting the insights attained through the above rigorous formalization and following the identified guidelines. While the asymptotic complexity analysis cannot expose the practical benefits due to worst-case assumptions, a series of experiments will demonstrate that this new learning algorithm outperforms virtually every existing one, in particular in the presence of non-minimal counterexamples.
- **Extension to richer classes:** The identified concepts and principles of an efficient algorithm for actively inferring DFAs is transferred to the setting of *visibly pushdown automata*, which can be used to model programs with recursion. The practical performance evaluation of this algorithm is further witness to the claim that a solid formal basis is key to achieving efficiency and scalability in practice.

1.3.1. Comments on Individual Contributions

Section 3.3 of this thesis is partly based on the paper *An Abstract Framework for Counterexample Analysis in Active Automata Learning* [108]. I was the lead author of all sections of this paper. The idea of applying other worst-case logarithmic search heuristics evolved in discussions with Bernhard Steffen. I was solely responsible for the formalization, implementation and for carrying out the experiments.

The version presented in this thesis differs from the framework presented in the above paper by allowing arbitrary (instead of binary) effect domains, which allows for instantiating the

framework in settings without unique representatives, and increases the efficiency when learning Mealy machines.

Chapter 5 is partly based on the paper *The TTT Algorithm: A Redundancy-free Approach to Active Automata Learning* [110]. I am the lead author of all sections of this paper. The idea of maintaining both prefix-closedness and suffix-closedness in a discrimination tree-based learning algorithm, which allows for storing the data in three trees, evolved in discussions among the authors of this paper. I was solely repsonsible for the algorithmic realization and working out the technical details, including in particular the realization of discriminator finalization, as well as the proof of space optimality. Furthermore, I was solely responsible for the implementation and conducting the experiments.

The version of TTT presented in this thesis differs from the description in the above paper in several aspects. First, it is now clearly specified that a step of counterexample analysis is only performed when no finalization is possible. Second, the finalized discriminator is obtained as the LCA of the successors of all states within a block, not just two arbitrary states, which allows to preserve semantic suffix-closedness. Third, the description has been adapted to show that soft sifting is sufficient for evaluating state output functions, which reduces the number of hard sifts required for counterexample analysis. In addition to the improved efficiency, this results in a much clearer specification of the algorithm. The evaluation in Section 5.5 uses an improved implementation based on the description in this thesis, and not the implementation that was used in the above paper. Again, I was solely responsible for all these extensions.

All other contents of this thesis, including the above-described extensions to the respective papers, are my own and original work, unless explicitly stated otherwise through citations.

1.4. Outline

This thesis is structured as follows: Chapter 2 establishes the notation used in this thesis, and provides definitions for frequently used mathematical concepts. It furthermore gives a brief overview on finite-state machines. These are discussed in much greater detail in Chapter 3, which also formally introduces the problem of active automata learning and describes the characteristics of deterministic finite automata that make learning them feasible in the first place. Based on this initial considerations, a mathematical framework for active automata learning algorithms is developed that will serve as the basis for algorithms developed in the remainder of the thesis.

The next two chapters are devoted to a detailed presentation of the TTT algorithm. Chapter 4 describes the data structure of *discrimination trees*, which play an essential role for efficient active automata learning due to their inherent redundancy freeness. For a clearer exposition of their characteristics, they are first presented in a white-box scenario, before describing how to realize black-box learning. Chapter 5 then describes the actual and technically very involved TTT algorithm, including a practical evaluation and comparison to other, previously existing algorithms.

Chapter 6 goes a step further, first describing how visibly-pushdown systems can be learned in a black-box setting. The second part of the chapter then describes how the ideas behind TTT can be transferred to this modified setting, resulting once more in a highly efficient algorithm.

Finally, Chapter 7 gives an overview on other works that are related to the topics of this thesis,

before Chapter 8 concludes the thesis, summing up its contents and discussing possible directions for future research.

2. Preliminaries

The aim of this chapter is to establish a common syntax and semantics for concepts that are relevant for this entire thesis. In particular, while it should be possible to only selectively read certain chapters of this thesis, the definitions and notations presented in this chapter are essential for almost all of them, and thus should not be skipped.

Conceptually, this chapter is divided into two sections: the first one focuses on purely mathematical concepts like functions, relations etc. While the reader is expected to have some basic understanding of these, they often appear with slight semantical variations in the literature (for example, is 0 an element of $\mathbb{N}$ or not?). Establishing a homogeneous and consistent syntax and semantics is thus the goal of this first section, along with introducing some more "exotic" notation, e.g., concerning partial functions.

The second section focuses on words and automata, both of which are structures that can be described in mathematical terms, but are often used with a distinct and well-established notation in the context of theoretical computer science.

Most of the definitions presented in this chapter are folklore and can be found in the same or similar ways in a large number of works of other authors. For additional information on the subject of automata theory and transition systems, we refer the reader to the standard literature [24, 91].

2.1. Mathematical Notation

The goal of this section is to introduce the notation for common concepts in mathematics that are of importance for this thesis. Of course, a mathematical background is indispensable for reading this thesis, as clearly not every single elementary concept can be introduced. For this reason, the description is limited to concepts where either no or several concurrently used definitions and notations exist.

2.1.1. Sets

Let $\mathbb{N}$ denote the set of non-negative integers (or natural numbers), including 0 (i.e., $\mathbb{N} = \{0,1,2,\ldots\}$). The set of positive integers is denoted by $\mathbb{N}^+$, while $\mathbb{Z}$ is the set of all integers (including negative ones). We furthermore define $\mathbb{B} =_{df} \{0,1\}$ as the set of *Boolean values*, where 0 is identified with *false* and 1 is identified with *true*. However, the values 0 and 1 will always be introduced explicitly in their respective contexts, and are not implicitly identified with the evaluation of some first-order logical statement such as $x \in X$.

For a set X, $|X|$ denotes its *cardinality*, i.e., the number of elements it contains. Furthermore, 2^X denotes the powerset of X, thus $2^X = \{X' \mid X' \subseteq X\}$.

2.1.2. Partial Functions

Let X and Y be arbitrary sets. A *partial function* f from X to Y, denoted by $f: X \rightharpoonup Y$, is a right-unique relation $f \subseteq X \times Y$. We write $f(x) = y$ if there exists $(x, y) \in f$ and say that $f(x)$ is *defined*; otherwise, we say that $f(x)$ is *undefined*. The *domain* of a partial function, denoted by $\mathrm{dom}\, f$, is the set of all $x \in X$ such that $f(x)$ is defined. Two partial functions $f_1, f_2: X \rightharpoonup Y$ are equal, denoted by $f_1 = f_2$, if and only if $\mathrm{dom}\, f_1 = \mathrm{dom}\, f_2$ and, for all $x \in \mathrm{dom}\, f_1$, $f_1(x) = f_2(x)$.

2.1.3. Equivalence Relations

A reflexive, symmetric and transitive binary relation $\approx \subseteq X \times X$ on an arbitrary set X is called an *equivalence relation* (on X). For $x \in X$, $[x]_\approx =_{df} \{x' \in X \mid x \approx x'\}$ denotes the *equivalence class of* x (with respect to $\approx$). An equivalence relation $\approx$ on X is said to *saturate* a subset $X' \subseteq X$ if and only if X' is the union of some equivalence classes of $\approx$. Note that in this case, we have

$$X' = \bigcup_{x \in X'} [x]_\approx,$$

and each equivalence class $[x]_\approx$ of $\approx$ is either a subset of or disjoint from X'. The *quotient* (or *quotient set*) of X with respect to an equivalence relation $\approx$ is defined as the set of all equivalence classes, and is denoted by $X/\approx =_{df} \{[x]_\approx \mid x \in X\}$. The *index* of an equivalence relation $\approx$, $\mathrm{ind}(\approx)$, is defined as the number of equivalence classes, i.e., $\mathrm{ind}(\approx) =_{df} |X/\approx|$.

The quotient forms a *partition* of X, and is also called the *partition of X induced by* $\approx$. In general, a partition of X is a set $P \subset 2^X$ such that (i) $\forall B \in P : B \neq \emptyset$, (ii) $\forall B, B' \in P : (B = B' \vee B \cap B' = \emptyset)$, and (iii) $\bigcup_{B \in P} B = X$. The elements of P are also called *blocks*. Each element $x \in X$ corresponds to exactly one block $B \in P$ such that $x \in B$. The *size* $|P|$ of a partition P is the number of distinct blocks it contains. If $|P| = k$, then P is also called a k-partition of X. If all elements of P are singletons (i.e., $|P| = |X|$ if X is finite), P is called the *discrete partition* of X. Just as an equivalence relation $\approx$ on X induces a partition of X, each partition $P \subset 2^X$ of X induces an equivalence relation $\sim_P \subseteq X \times X$, such that $x \sim_P x'$ if and only if x and x' are in the same block of P.

For an arbitrary function $f: X \to Y$ mapping elements of X to some arbitrary set Y, $\sim_f \subseteq X \times X$ denotes the *equivalence kernel of* f, which is defined via $x_1 \sim_f x_2 \Leftrightarrow_{df} f(x_1) = f(x_2)$. For simplicity, we denote the equivalence class of x with respect to $\sim_f$ by $[x]_f$, instead of the more explicit $[x]_{\sim_f}$. The quotient $X/\sim_f$ is also referred to as the *partition (of X) induced by* f.

Given two equivalence relations $\approx_1, \approx_2 \subseteq X \times X$ on X, $\approx_2$ is said to *refine* $\approx_1$ if and only if for all $x, x' \in X$, $x \approx_2 x'$ implies $x \approx_1 x'$. In this case, each equivalence class of $\approx_1$ is a (disjoint) union of equivalence classes of $\approx_2$. Moreover, for the cardinality of the quotient sets, we have $|X/\approx_2| \geq |X/\approx_1|$. Note that each equivalence relation refines itself; if $\approx_2$ and $\approx_1$ are distinct in the above case, we say that the refinement is *strict* (or, that $\approx_2$ *strictly refines* $\approx_1$). If both the refinement is strict *and* the quotient set $|X/\approx_1|$ is finite, the above property can be strengthened to $|X/\approx_2| > |X/\approx_1|$.

2.2. Alphabets, Words, and Automata

Automata, or finite-state machines, are an important concept in theoretical computer science, used for modeling a large class of systems. In this thesis, we are primarily concerned with finite-state machines, that furthermore operate on a finite input alphabet.

2.2.1. Alphabets and Words

Throughout this thesis, let Σ be an arbitrary non-empty[1] *alphabet*. A finite sequence of elements of Σ (which in this context are called *symbols*) is called a (finite) *word* over Σ.[2] In the following, we fix the alphabet Σ, and will omit the explicit "over Σ" when talking about words.

The length of a word w is defined as the length of this sequence, and is denoted by $|w|$. The unique word of length zero is called the *empty word*, and is denoted by ε. Single symbols $a \in \Sigma$ are identified with words of length 1. We write Σ^m for the set of all words of length $m \in \mathbb{N}$, and $\Sigma^{\leq m}$ for the set of all words of length up to m, i.e.,

$$\Sigma^{\leq m} =_{df} \bigcup_{i=0}^{m} \Sigma^i.$$

The set of all words of arbitrary (but finite) length is denoted by Σ^*, and Σ^+ denotes the set of all *non-empty* words. These can be defined as

$$\Sigma^* =_{df} \bigcup_{i=0}^{\infty} \Sigma^i \quad \text{and} \quad \Sigma^+ =_{df} \bigcup_{i=1}^{\infty} \Sigma^i,$$

respectively. Note that $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Let $w \in \Sigma^*$ be a word of length $m \in \mathbb{N}$, and assume that $w_1, \ldots, w_m \in \Sigma$ are the single symbols of which w consists; we also write $w = w_1 \ldots w_m$ to express this fact.[3] The *concatenation* of words $w, w' \in \Sigma^*$, denoted by $w \cdot w'$, is the word obtained from concatenating the symbol sequences of w and w'. Thus, if $w = w_1 \ldots w_m$ and $w' = w'_1 \ldots w'_{m'}$, $w \cdot w' = w_1 \ldots w_m w'_1 \ldots w'_{m'}$ and $|w \cdot w'| = m + m'$. Concatenation is an associative operation, meaning that for $w, w', w'' \in \Sigma^*$, we have $(w \cdot w') \cdot w'' = w \cdot (w' \cdot w'') = w \cdot w' \cdot w''$. We will sometimes omit the "$\cdot$" symbol, and simply write $w\,w'$ for the concatenation of w and w'. An explicit "$\cdot$" will be written either to improve readability, or to emphasize a logical subdivision of the concatenated word. We lift the concatenation operation to sets of words in the natural way: for $U, V \subseteq \Sigma^*$, we have

$$U \cdot V =_{df} \{u \cdot v \mid u \in U, v \in V\}.$$

We furthermore allow either of the operands of this lifted concatenation operation to be a single word instead of a set, which is then identified with the corresponding singleton set. That is, for $u \in \Sigma^*$ and $V \subseteq \Sigma^*$, we have $u \cdot V = \{u\} \cdot V$.

For $U \subseteq \Sigma^*$ and $i \in \mathbb{N}$, U^i denotes the set containing all words that can be represented by concatenating i (not necessarily distinct) words from U, i.e., $U^0 =_{df} \{\varepsilon\}$, and $U^{i+1} =_{df} U \cdot U^i$ for $i \in \mathbb{N}$. Similarly to the definition of Σ^*, the *Kleene star* operation on a set U is defined as

$$U^* =_{df} \bigcup_{i=0}^{\infty} U^i.$$

[1] While it is possible to define some of the following concepts for empty alphabets also, we will generally assume alphabets to be non-empty, unless explicitly stated otherwise.

[2] In the literature, the term *string* is frequently used in lieu of *word*. However, we prefer the latter, as *string* in a computer science context is commonly associated with being defined over some "natural" alphabet, such as the set of all UTF-16 characters.

[3] In certain circumstances, the need for introducing finite sequences of words will arise, the elements of which might also be named w_i. This should however not lead to any confusion: whenever a word is introduced as, e.g., $w \in \Sigma^*$, w_i refers to the i-th symbol of w. Otherwise, if w_i refers to a word in a sequence of words, we explicitly state $w_i \in \Sigma^*$.

A *subword* of a word $w \in \Sigma^*$ is a word $w' \in \Sigma^*$ such that there exist $u, v \in \Sigma^*$ satisfying $w = u \cdot w' \cdot v$. Assuming that $w = w_1 \dots w_m$ is a word of length $m = |w|$, for $1 \leq i \leq j \leq m$, $w_{i..j}$ denotes the subword $w_i \dots w_j$ of w. If $i > j$, then $w_{i..j} = \varepsilon$.[4]

A *prefix* of a word $w \in \Sigma^*$ is a word $u \in \Sigma^*$ such that there exists a word $v \in \Sigma^*$ satisfying $w = u \cdot v$, i.e., $u = w_{1..i}$ for some $0 \leq i \leq |w|$. We write $u \sqsubseteq_{pref} w$ to express the fact that u is a prefix of w. If furthermore $|u| < |w|$ (i.e., $u \neq w$) holds, u is called a *strict prefix* of w. The *prefix set* of a word w, i.e., the set of all its prefixes, is denoted by $\mathrm{Pref}(w) =_{df} \left\{ u \in \Sigma^* \mid u \sqsubseteq_{pref} w \right\}$. This definition can be generalized to sets of words $S \subseteq \Sigma^*$ in the following, natural way:

$$\mathrm{Pref}(S) =_{df} \bigcup_{w \in S} \mathrm{Pref}(w).$$

S is called *prefix-closed* if and only if $\mathrm{Pref}(S) = S$.

The counterpart of a prefix is a *suffix*. Formally, $v \in \Sigma^*$ is a suffix of $w \in \Sigma^*$, denoted by $v \sqsubseteq_{suff} w$, if there exists a word $u \in \Sigma^*$ such that $w = u \cdot v$ (i.e., $v = w_{i..|w|}$ for some $1 \leq i \leq m+1$). The concepts of *suffix set*, *suffix-closedness*, and *strict suffix* are defined in analogy to their prefix counterparts. Note that both $\sqsubseteq_{pref} \subseteq \Sigma^* \times \Sigma^*$ and $\sqsubseteq_{suff} \subseteq \Sigma^* \times \Sigma^*$ are partial orders on the set Σ^*.

For $u \in \Sigma^*$ and an arbitrary $v \in \Sigma^*$, the word $u \cdot v$ is called an *extension* of u. The set of all extensions of a word, $u\Sigma^*$, is thus the largest set such that u is a prefix of all its elements. In the special case that $|v| = 1$, $u \cdot v$ is called a *one-letter extension*. The (finite, if Σ is finite) set of all possible one-letter extensions is denoted by $u\Sigma$.

2.2.2. Transition Systems

Transition systems are a ubiquitous concept in computer science, as they describe, in an abstract way, the evolution of a system over time (or an abstracted, often discrete version thereof). This evolution is described by changes in the *state* of such a system. In the most general sense, the concept of a state merely encompasses the potential future evolutions of the system, though usually some context-dependent interpretation is attached to single states (such as it being "accepting" or "rejecting" in the context of finite-state acceptors, see below). A *transition* from one state to the next (successor state) is often associated with some label ("action"), which may be associated with some externally triggered event (e.g., a button being pressed, network data being received), but can also correspond to an implicit event such as a certain period of time having passed.

The notion of transition systems that we will introduce here will merely serve syntactical purposes, that is, establishing a common notation for reasoning about (specific) evolutions of states in such a system. Thus, a transition system in the following sense usually does not occur explicitly as part of some problem or as input to an algorithm, but rather is a structure induced by some other object such as a DFA, as described in the next subsection.

[4] This also applies if j is an otherwise invalid index, such as $i = 1, j = 0$.

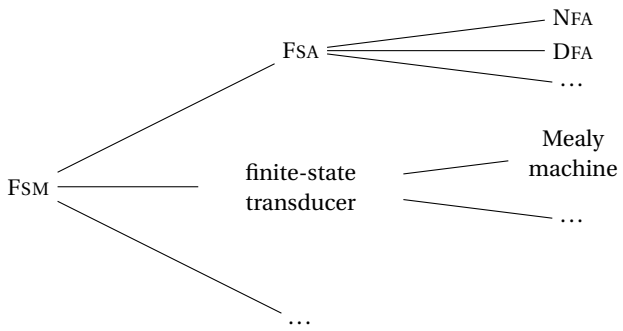


Figure 2.1.: Taxonomy of various types of finite-state machines

Definition 2.1 (*Transition system*)

A *transition system* is a triple $\langle \mathcal{S}, Act, \rightarrow \rangle$, where

- $\mathcal{S}$ is a set of *states*,
- Act is a set of *actions*,
- $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the *transition relation*.

For states $s, s' \in \mathcal{S}$ and an action $a \in Act$, we write $s \xrightarrow{a} s'$ to denote that $(s,a,s') \in \rightarrow$. $s \longrightarrow s'$ expresses that there exists some $a \in Act$ such that $s \xrightarrow{a} s'$. For a sequence (or word) of actions $w = a_1 a_2 \ldots a_m \in Act^*$, we write $s \overset{w}{\Rightarrow} s'$ if there exist states $s_0, \ldots, s_m \in \mathcal{S}$ such that $s_0 = s$, $s_m = s'$, and $s_{i-1} \xrightarrow{a_i} s_i$ for all $1 \leq i \leq m$ (note that if $w = \varepsilon$ and thus $m = 0$, this reduces to $s = s'$). Again, $s \Rightarrow s'$ denotes that there exists some $w \in Act^*$ such that $s \overset{w}{\Rightarrow} s'$.

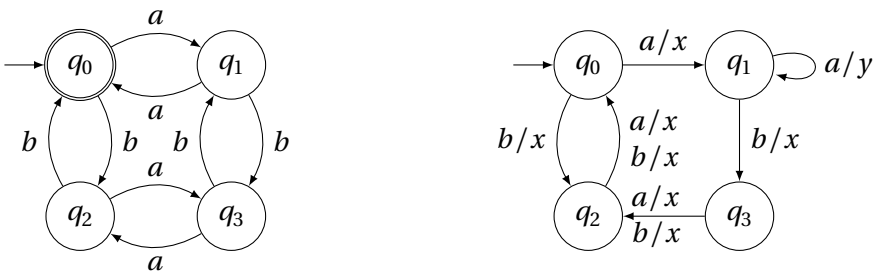
2.2.3. Finite-State Acceptors

Transition systems are often given through one of the various forms of *finite-state machines* (FSMs). Intuitively, an FSM gives rise to a transition system over a *finite* set of states (usually denoted by Q instead of $\mathcal{S}$), where each transition is triggered by an action from a *finite* set of actions (denoted by Σ instead of Act, and referred to as the *alphabet*). Apart from this common property of finite state-space and alphabet, there exists a large variety of different finite-state machine models for various tasks, e.g., realizing a binary classifier for words, or translating input words into output words (over a potentially different alphabet). Figure 2.1 visualizes a fragment of a *taxonomy* of different types of FSMs, including those that are considered in this thesis. The next sections will formally introduce the mentioned machine models.

Finite-state acceptors (FSAs) are certainly among the most fundamental concepts in theoretical computer science. Conceptually, they realize a binary classifier for (finite) words over some finite alphabet Σ, i.e., they can be seen as computing a unary predicate on Σ^*.

FSAs generally come in two flavors: *non-deterministic finite automata* (NFAs) and *deterministic finite automata* (DFAs).[5] While DFAs have a lot of desirable properties and are conceptually

[5]The term "(non-)deterministic finite automaton" is well-established for these kinds of finite-state acceptors, although it could be criticized that *finite-state acceptor* would be a better substitute for *finite automaton*, due to the vagueness of the latter.



(a) Example NFA (that is also a DFA)

(b) Example Mealy machine

Figure 2.2.: Example FSM visualizations

much simpler to work with, we will first introduce NFAs, as they form the general case.

Definition 2.2 (NFA)

Let Σ be a finite input alphabet. A *non-deterministic finite automaton* (NFA) $\mathcal{A}$ (over Σ) is a tuple $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, Q_{0,\mathcal{A}}, \Delta_\mathcal{A}, F_\mathcal{A} \rangle$, where

- $Q_\mathcal{A}$ is a finite, non-empty set of *states*,
- $Q_{0,\mathcal{A}} \subseteq Q_\mathcal{A}$ is a non-empty set of *initial states*,
- $\Delta_\mathcal{A} \subseteq Q_\mathcal{A} \times \Sigma \times Q_\mathcal{A}$ is a *transition relation*, where $(q,a,q') \in \Delta_\mathcal{A}$ indicates that the automaton can move from state q to state q' upon reading the input symbol a, and
- $F_\mathcal{A} \subseteq Q_\mathcal{A}$ is a set of *final* (or *accepting*) states.

Semantics of an NFA. An NFA $\mathcal{A}$ induces the transition system $\langle \mathcal{S}, Act, \rightarrow \rangle$ with $\mathcal{S} = Q_\mathcal{A}$, $Act = \Sigma$, and $\rightarrow = \Delta_\mathcal{A}$. A word $w \in \Sigma^*$ is said to be *accepted* by $\mathcal{A}$ if and only if there exists an initial state $q \in Q_{0,\mathcal{A}}$ and an accepting state $q' \in F_\mathcal{A}$ such that $q \overset{w}{\Rightarrow} q'$, otherwise it is said to be *rejected* by $\mathcal{A}$. The *language* $\mathcal{L}(\mathcal{A})$ of an NFA $\mathcal{A}$ (or the language *accepted* by $\mathcal{A}$) is the set of all accepted words, i.e., $\mathcal{L}(\mathcal{A}) =_{df} \left\{ w \in \Sigma^* \mid \exists q \in Q_{0,\mathcal{A}}, q' \in F_\mathcal{A} : q \overset{w}{\Rightarrow} q' \right\}$.

In many contexts, it is more convenient to refer to the semantics of an NFA $\mathcal{A}$ in terms of a *function* instead of its language, which is a set. In fact, this is indispensable for the case of *finite-state transducers* (see Section 2.2.4), and thus a prerequisite for a generalization of the theory developed in the next chapter.

Definition 2.3 (*Output function*)

Let $\mathcal{A}$ be an NFA over an input alphabet Σ. The *output function* of $\mathcal{A}$, $\lambda_\mathcal{A}$, is defined as

$$\lambda_\mathcal{A} : \Sigma^* \rightarrow \mathbb{B}, \quad \lambda_\mathcal{A}(w) =_{df} \begin{cases} 1 \text{ if } w \in \mathcal{L}(\mathcal{A}) \\ 0 \text{ otherwise} \end{cases} \qquad \forall w \in \Sigma^*.$$

Note that $\lambda_\mathcal{A}$ is simply the *characteristic function* (or *indicator function*) of $\mathcal{L}(\mathcal{A})$. Throughout this entire thesis, we will prefer output functions over languages: for a language $L \subseteq \Sigma^*$, we will

refer to its characteristic function as its *output function* $\lambda_L : \Sigma^* \to \mathbb{B}$. Thus, $\lambda_{\mathcal{A}} = \lambda_{\mathcal{L}(\mathcal{A})}$. Conversely, for an arbitrary output function $\lambda : \Sigma^* \to \mathbb{B}$, the corresponding language is $\lambda^{-1}(1)$.

Visualization of NFAs. NFAs (or, FSMs in general) are typically visualized by representing their transition system as a graph: nodes correspond to states (drawn as circles), and edges (drawn as arrows) between nodes indicate the existence of a transition. The edges are typically labeled with the corresponding action from Σ. The initial states are visualized by having an incoming edge with no source node and no label. In the case of NFAs, states can furthermore be accepting or rejecting. This is commonly visualized by drawing the accepting states with a double circle, and the rejecting ones with a single circle.

Figure 2.2a shows an NFA that recognizes the language over $\Sigma = \{a, b\}$ containing an even number of as and bs. It has four states, q_0 through q_3, where q_0 is the only initial and also the only accepting state.

Remark 2.1

In Definition 2.2, the transitions of an NFA $\mathcal{A}$ are described in terms of a *transition relation* $\Delta_{\mathcal{A}}$. However, it can be useful to treat this relation as a *function* of a state and an input symbol, mapping into the powerset of $Q_{\mathcal{A}}$. Thus, the (non-deterministic) *transition function* $\delta_{\mathcal{A}} : Q_{\mathcal{A}} \times \Sigma \to 2^{Q_{\mathcal{A}}}$ is defined as:

$$\delta_{\mathcal{A}}(q,a) =_{df} \left\{ q' \in Q_{\mathcal{A}} \mid (q,a,q') \in \Delta_{\mathcal{A}} \right\} \quad \forall q \in Q_{\mathcal{A}}, a \in \Sigma.$$

We first lift $\delta_{\mathcal{A}}$ to *sets* of states in the usual fashion, i.e.,

$$\delta_{\mathcal{A}}(Q',a) =_{df} \bigcup_{q \in Q'} \delta_{\mathcal{A}}(q,a) \quad \forall Q' \subseteq Q_{\mathcal{A}}, a \in \Sigma,$$

and then use this to define the extension of $\delta_{\mathcal{A}}$ to words $w \in \Sigma^*$, denoted by $\delta^*_{\mathcal{A}} : Q_{\mathcal{A}} \times \Sigma^* \to 2^{Q_{\mathcal{A}}}$, in the following, inductive fashion:

$$\begin{array}{lll} \delta^*_{\mathcal{A}}(q,\varepsilon) & =_{df} \{q\} & \forall q \in Q_{\mathcal{A}}, \\ \delta^*_{\mathcal{A}}(q,a \cdot w) & =_{df} \delta^*_{\mathcal{A}}(\delta_{\mathcal{A}}(q,a),w) & \forall q \in Q_{\mathcal{A}}, a \in \Sigma, w \in \Sigma^*. \end{array}$$

Note that $\delta^*_{\mathcal{A}}$ could alternatively be defined in terms of the relation $\Rightarrow$ defined in Section 2.2.2: we have $\delta^*_{\mathcal{A}}(q,w) = \left\{ q' \in Q_{\mathcal{A}} \mid q \overset{w}{\Rightarrow} q' \right\}$ for all $q \in Q_{\mathcal{A}}, w \in \Sigma^*$.

We will furthermore follow the common approach of identifying $\delta_{\mathcal{A}}$ and $\delta^*_{\mathcal{A}}$, motivated by the fact that they coincide for arguments of length 1. Hence, in the remainder, $\delta_{\mathcal{A}}$ can refer to both the "normal" as well as the extended transition function of $\mathcal{A}$.

The above remark on treating the transition relation as a transition function allows us to conveniently define two important properties of NFAs.

Definition 2.4 (*determinism, completeness*)

Let $\mathcal{A}$ be an NFA over Σ. $\mathcal{A}$ is called:

(i) *deterministic*[6] if and only if $|Q_{0,\mathcal{A}}| = 1$ and $|\delta_{\mathcal{A}}(q,a)| \leq 1$ for all $q \in Q_{\mathcal{A}}, a \in \Sigma$.

(ii) *complete* if and only if $\delta_{\mathcal{A}}(q,a) \neq \emptyset$ for all $q \in Q_{\mathcal{A}}, a \in \Sigma$.

Deterministic and complete NFAs are of such importance that it is common to define them in their own, slightly adjusted fashion, instead of treating them as restricted NFAs. In particular, replacing the *set* of initial states $Q_{0,\mathcal{A}}$ with a single initial state $q_{0,\mathcal{A}} \in Q_{\mathcal{A}}$, and the transition relation $\Delta_{\mathcal{A}}$ with a *deterministic* transition function $\delta_{\mathcal{A}} \colon Q_{\mathcal{A}} \times \Sigma \to Q_{\mathcal{A}}$ in Definition 2.2, one arrives at the following, common definition of *deterministic finite automata* (DFA).

Definition 2.5 (DFA)

Let Σ be a finite input alphabet. A *deterministic finite automaton* (DFA) $\mathcal{A}$ is a tuple $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, q_{0,\mathcal{A}}, \delta_{\mathcal{A}}, F_{\mathcal{A}} \rangle$, where

- $Q_{\mathcal{A}}$ is a finite, non-empty set of *states*,
- $q_{0,\mathcal{A}} \in Q_{\mathcal{A}}$ is the designated *initial state*,
- $\delta_{\mathcal{A}} \colon Q_{\mathcal{A}} \times \Sigma \to Q_{\mathcal{A}}$ is the *transition function*, where $\delta_{\mathcal{A}}(q,a) = q'$ indicates that $\mathcal{A}$ moves from state q to state q' upon reading input symbol a, and
- $F_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$ is the set of *final* (or *accepting*) states.

As each DFA $\mathcal{A}$ is also an NFA,[7] we do not need to re-define the formal semantics for DFAs, nor specify how they are visualized (incidentally, the NFA from Figure 2.2a is also a DFA). It should be noted that the transition function $\delta_{\mathcal{A}}$ maintains its functional nature when extended to words (cf. Remark 2.1). The definition of the language recognized by a DFA $\mathcal{A}$ can thus be rephrased slightly more concisely as

$$\mathcal{L}(\mathcal{A}) =_{df} \left\{ w \in \Sigma^* \mid \delta_{\mathcal{A}}(q_{0,\mathcal{A}}, w) \in F_{\mathcal{A}} \right\}.$$

The output function $\lambda_{\mathcal{A}}$ for a DFA is defined in the same way as for NFAs, and we therefore also have $\mathcal{L}(\mathcal{A}) = \lambda_{\mathcal{A}}^{-1}(1)$. However, particularly in the context of DFAs it is convenient to also define an output function for individual states.

[6] The fact that *non-deterministic* finite automata can be *deterministic* may sound confusing at first. However, while the latter refers to a property that may be present in instances of the formalism, the former describes a degree of freedom that the formalism permits, not enforces.

[7] There is a slight syntactical difference in the transition function $\delta_{\mathcal{A}}$ when $\mathcal{A}$ is treated as a DFA versus when it is treated as an NFA: in the former case, it maps into $Q_{\mathcal{A}}$, whereas in the latter case it maps into $2^{Q_{\mathcal{A}}}$. However, due to determinism and completeness (Definition 2.4), the images of the NFA-style transition function are guaranteed to be singleton sets, which we identify with their only element.

Definition 2.6 (*State output function*)

Let $\mathcal{A}$ be a DFA over Σ, and let $q \in Q_\mathcal{A}$ be an arbitrary state of $\mathcal{A}$. The *state output function* of $q \in Q_\mathcal{A}$, $\lambda^q_\mathcal{A}$, is defined as

$$\lambda^q_\mathcal{A} : \Sigma^* \to \mathbb{B}, \quad \lambda^q_\mathcal{A}(w) =_{df} \begin{cases} 1 \text{ if } \delta_\mathcal{A}(q,w) \in F_\mathcal{A} \\ 0 \text{ otherwise} \end{cases} \quad \forall w \in \Sigma^*.$$

Note that the state output function $\lambda^q_\mathcal{A}$ is essentially the output function $\lambda_{\mathcal{A}'}$ of a DFA $\mathcal{A}$ that is derived from $\mathcal{A}$ by changing the initial state to q. Thus, $\lambda_\mathcal{A} = \lambda^{q_{0,\mathcal{A}}}_\mathcal{A}$.

2.2.4. Finite-State Transducers

A DFA can either accept or reject an input word, i.e., it computes a Boolean-valued output function $\lambda : \Sigma^* \to \mathbb{B}$. The restriction of the codomain to $\mathbb{B}$ is often inadequate for modeling the output behavior of realistic systems. In the general case, the output behavior is described by a function $\lambda : \Sigma^* \to \mathcal{D}$, where $\mathcal{D}$ is some arbitrary *output domain*. *Reactive systems* [128], which usually do not terminate but only produce *intermediate* outputs, obey even further restrictions, as will be discussed below.

It is typical to choose $\mathcal{D} = \Omega^*$, for some *output alphabet* Ω. The output function $\lambda : \Sigma^* \to \Omega^*$ thus takes words over one alphabet, Σ, and translates them into words over another alphabet, Ω. Machines that accomplish this task are commonly referred to as *transducers*, and for the special case that they compute the output function in finite space, as *finite-state transducers*.

There are many formalisms for finite-state transducers. We will concentrate on the particularly simple and widely used one of a *Mealy machine*.

Definition 2.7 (*Mealy machine*)

Let Σ be a finite input alphabet and Ω be a finite output alphabet. A *Mealy machine* over Σ and Ω is a tuple $\mathcal{M} = \langle Q_\mathcal{M}, \Sigma, \Omega, q_{0,\mathcal{M}}, \delta_\mathcal{M}, \gamma_\mathcal{M} \rangle$, where

- $Q_\mathcal{M}$ is a finite, non-empty set of *states*,
- $q_{0,\mathcal{M}} \in Q_\mathcal{M}$ is the designated *initial state*,
- $\delta_\mathcal{M} : Q_\mathcal{M} \times \Sigma \to Q_\mathcal{M}$ is the *transition function*, and
- $\gamma_\mathcal{M} : Q_\mathcal{M} \times \Sigma \to \Omega$ is the *transition output function*.

Semantics of a Mealy machine. Upon reading an input symbol $a \in \Sigma$, a Mealy machine $\mathcal{M}$ moves from the current state $q \in Q_\mathcal{M}$ (starting with the initial state) to the successor state $\delta_\mathcal{M}(q,i)$ while producing the output symbol $\gamma_\mathcal{M}(q,a)$. The concatenation of all output symbols that have been produced when reading an input word $w \in \Sigma^*$ forms the *output* of $\mathcal{M}$ in response to w. Thus, a Mealy machine $\mathcal{M}$ computes an output function $\lambda_\mathcal{M} : \Sigma^* \to \Omega^*$.

To give a formal definition of $\lambda_\mathcal{M}$, let us first introduce the *extended transition output function* $\gamma^*_\mathcal{M} : Q_\mathcal{A} \times \Sigma^* \to \Omega^*$, which—in analogy to the extended transition function (cf. Remark 2.1)—extends the normal transition output function $\gamma_\mathcal{M} : Q_\mathcal{A} \times \Sigma \to \Omega$ from single symbols to words. It

is defined inductively as follows:

$$\begin{aligned} \gamma^*_{\mathcal{M}}(q,\varepsilon) &=_{df} \varepsilon, \\ \gamma^*_{\mathcal{M}}(q,a\cdot w) &=_{df} \gamma_{\mathcal{M}}(q,a)\cdot\gamma^*_{\mathcal{M}}(\delta_{\mathcal{M}}(q,a),w) \;\; \forall q\in Q_{\mathcal{M}}, w\in\Sigma^*, a\in\Sigma. \end{aligned}$$

Again, we will identify $\gamma_{\mathcal{M}}$ and $\gamma^*_{\mathcal{M}}$, as the latter coincides with the former for arguments of length one. The output function $\lambda_{\mathcal{M}}$ can then simply be defined via $\lambda_{\mathcal{M}}(w) =_{df} \gamma_{\mathcal{M}}(q_{0,\mathcal{M}}, w)$. Note that also for a Mealy machine $\mathcal{M}$ we can define a *state output function* (cf. Definition 2.6): for $q\in Q_{\mathcal{M}}$, the state output function $\lambda^q_{\mathcal{M}}\colon \Sigma^*\to\Omega^*$ is simply defined as $\lambda^q_{\mathcal{M}}(w) =_{df} \gamma_{\mathcal{M}}(q,w)$, for all $w\in\Sigma^*$.

Remark 2.2

The output function of a Mealy machine $\mathcal{M}$ will always satisfy the following two properties:

$$\forall w,w'\in\Sigma^*: w\sqsubseteq_{pref} w' \Rightarrow \lambda_{\mathcal{M}}(w)\sqsubseteq_{pref}\lambda_{\mathcal{M}}(w') \tag{2.1}$$

and

$$\forall w\in\Sigma^*: |\lambda_{\mathcal{M}}(w)| = |w|. \tag{2.2}$$

(2.1) corresponds to the property of a (deterministic) *reactive system* [128] that never terminates, but instead continuously receives inputs from the environment and produces output symbols. Thus, every output $\lambda_{\mathcal{M}}(w)$ in response to a finite input word $w\in\Sigma^*$ can always be extended by supplying new inputs, i.e., extending w.

The property (2.2) establishes a one-to-one correspondence between input and output symbols. Note that the output alphabet Ω may contain arbitrarily complex symbols, including a special symbol indicating no output (*quiescence*), or symbols corresponding to several outputs produced consecutively. However, it is impossible to directly model systems that produce outputs before receiving the first input, or systems that keep on producing outputs indefinitely, without receiving inputs in between. Such phenomena can only be modeled indirectly, e.g., by introducing special input symbols for initialization or for indicating the absence of an actual input.

Transducers satisfying (2.2) are sometimes also called *letter-to-letter transducers* (e.g., by Sakarovitch [158]). Letter-to-letter transducers are significantly simpler to infer in a *passive learning* setting (a survey on this topic, including passive inference of various kinds of transducers that are *not* letter-to-letter transducers, is given by de la Higuera [61]), as it is then not necessary to determine which *subwords* of the input and output words correspond to the same transition. In an active learning context, however, this *alignment* can be inferred trivially by querying $\lambda(w')$ for each $w'\in\mathrm{Pref}(w)\setminus\{\varepsilon\}$. As we are solely considering active learning in this thesis, we can thus neglect this difference, and will use the term "transducer" synonymously with "letter-to-letter transducer".

Visualizing Mealy machines. The transition structure of a Mealy machine $\mathcal{M}$ is visualized in a similar way as for NFAs and DFAs: states in $Q_{\mathcal{M}}$ are drawn as circles (note that for Mealy machines, there is no concept of acceptance, thus all states are drawn as single circles). The initial state has an unlabeled incoming edge that has no source state. An edge from state q to state q' labeled with a/o expresses the fact that $q' = \delta_{\mathcal{M}}(q,a)$ and $o = \gamma_{\mathcal{M}}(q,a)$. Figure 2.2b shows a Mealy machine over $\Sigma = \{a,b\}$ with four states, q_0 (the initial state) through q_3, and using $\Omega = \{x,y\}$ as its output alphabet.

2.2.5. Common FSM Concepts

To conclude this chapter, we want to discuss some important concepts and operations that uniformly apply to all the presented types of (deterministic) FSMs. The generalization of these concepts is essential for a theory of automata learning that is not inherently tied to a specific machine model such as DFAs. While it is inevitable that *some* distinctions need to be made at the model level (for example, it is at least not obvious how the concept of whether a state is accepting or not (DFAs) and what the output of a transition is (Mealy machines) could be treated uniformly), these specific algorithms can build upon a common basis that is formulated at what a computer engineer would call the *interface* level.

In the following, we will assume that a deterministic FSM $\mathcal{A}$ over Σ has states $Q_\mathcal{A}$, an initial state $q_{0,\mathcal{A}}$, a transition function $\delta_\mathcal{A}$, and an output function $\lambda_\mathcal{A}$. We further assume that the output function $\lambda_\mathcal{A}$ maps from Σ^* to some output domain $\mathcal{D}$ (i.e., $\mathcal{D}=\mathbb{B}$ for DFAs, and $\mathcal{D}=\Omega^*$ for Mealy machines), and that for every state $q\in Q_\mathcal{A}$, there exists a state output function $\lambda^q_\mathcal{A}\colon \Sigma^*\to\mathcal{D}$. In general, the state output function $\lambda^q_\mathcal{A}$ is the output function $\lambda_{\mathcal{A}'}$ of the FSM $\mathcal{A}'$ obtained from $\mathcal{A}$ by changing the initial state to $q_{0,\mathcal{A}'} =_{df} q$.

Reached and reachable states. The fact that a deterministic FSM has a designated initial state $q_{0,\mathcal{A}}\in Q_\mathcal{A}$, along with the functional characteristics of the (extended) transition function, ensures that for every word $w\in\Sigma^*$, there is a unique state q that is *reached* by w from the initial state. We will denote this state by $\mathcal{A}[w]$, thus $\mathcal{A}[w]=_{df}\delta_\mathcal{A}(q_{0,\mathcal{A}},w)$. A state $q\in Q_\mathcal{A}$ is called *reachable* if and only if there exists $w\in\Sigma^*$ such that $\mathcal{A}[w]=q$. The notation is extended to sets of words in the natural fashion, yielding

$$\mathcal{A}[W]=_{df}\left\{\mathcal{A}[w]\,|\,w\in\Sigma^*\right\}\quad\forall W\subseteq\Sigma^*.$$

The set of *reachable states* is thus $\mathcal{A}[\Sigma^*]$. If $Q_\mathcal{A}=\mathcal{A}[\Sigma^*]$ (i.e., all states are reachable), $\mathcal{A}$ is called *trim*. If an FSM $\mathcal{A}$ is not trim, i.e., the set of unreachable states $Q_\mathcal{A}\setminus\mathcal{A}[\Sigma^*]$ is nonempty, the FSM $\mathcal{A}'$ obtained by removing all unreachable states (that is, $Q_{\mathcal{A}'}=\mathcal{A}[\Sigma^*]$ and $\delta_{\mathcal{A}'}$ is obtained as the restriction of $\delta_\mathcal{A}$ to $Q_{\mathcal{A}'}\times\Sigma$) computes the same output function as $\mathcal{A}$.

Semantic equivalence and separators. The following definition formally states what it means when two FSMs $\mathcal{A}$ and $\mathcal{A}'$ are *semantically equivalent*.

Definition 2.8 (*Equivalent* FSM*s and states*)

(i) Let $\mathcal{A}$ and $\mathcal{A}'$ be FSMs. $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent* ($\mathcal{A}\equiv\mathcal{A}'$) if and only if their output functions are equal, i.e., $\lambda_\mathcal{A}=\lambda_{\mathcal{A}'}$.[8]

(ii) Let $\mathcal{A}$ be a deterministic FSM, and let $q,q'\in Q_\mathcal{A}$ be states of $\mathcal{A}$. q and q' are *equivalent* ($q\equiv q'$) if and only if their output functions are equal, i.e., $\lambda^q_\mathcal{A}=\lambda^{q'}_\mathcal{A}$.

Incidentally, two FSMs $\mathcal{A}$ and $\mathcal{A}'$ are equivalent if and only if their initial states are equivalent, i.e., $\mathcal{A}\equiv\mathcal{A}'\Leftrightarrow q_{0,\mathcal{A}}\equiv q_{0,\mathcal{A}'}$. To explicitly separate the equivalence between states from the equivalence between FSMs, we will also write $\equiv_\mathcal{A}$ to denote the equivalence between states of some FSM $\mathcal{A}$.

[8] Note that, for FSMs $\mathcal{A}$ and $\mathcal{A}'$ to be equivalent, at the very minimum they have to operate on a common input alphabet Σ, and their output images have to be equal, i.e., $\lambda_\mathcal{A}(\Sigma)=\lambda_{\mathcal{A}'}(\Sigma)$, implying that their output domains intersect.

Equality of functions over a common domain is defined pointwisely, i.e., for two functions to be equal they need to have the same domain, and need to map every element of this domain to the same value. Thus, if two (state) output functions are not equal (and thus the FSMs or states not equivalent), but have the same domain Σ^*, their values for at least one argument $w \in \Sigma^*$ must differ.

Definition 2.9 (*Separators*)

(i) Let $\mathcal{A}$ and $\mathcal{A}'$ be FSMs over Σ such that $\mathcal{A} \not\equiv \mathcal{A}'$. A *separator* (or *inequivalence witness*) for $\mathcal{A}$ and $\mathcal{A}'$ is a word $w \in \Sigma^*$ such that $\lambda_{\mathcal{A}}(w) \neq \lambda_{\mathcal{A}'}(w)$.

(ii) Let $q, q' \in Q_{\mathcal{A}}$ be states of a deterministic FSM $\mathcal{A}$ such that $q \not\equiv_{\mathcal{A}} q'$. A *separator* (or *inequivalence witness*) for q and q' is a word $w \in \Sigma^*$ such that $\lambda^q_{\mathcal{A}}(w) \neq \lambda^{q'}_{\mathcal{A}}(w)$.

Two FSMs or states which are inequivalent (i.e., for which there exists a separator) are thus also called *separable*. There typically exist (often infinitely) many different separators for two FSMs or states. For example, in the case of Mealy machines, every extension of a separator is again a separator. This motivates a minimality criterion for separators: a separator is *minimal* if none of its strict prefixes is a separator. Again, there may exist a large (or infinite) number of minimal separators, and two minimal separators may differ vastly in their length. There may furthermore be several different *shortest* separators (i.e., separators of minimum length).

Minimality, isomorphisms and canonicity. There often is a large number of different FSMs realizing the same output function $\lambda : \Sigma^* \to \mathcal{D}$, all of which are equivalent to each other. Of particular interest among these are the ones with a minimum number of states. Precisely, an FSM $\mathcal{A}$ is called *minimal* if every other FSM $\mathcal{A}'$ such that $\mathcal{A} \equiv \mathcal{A}'$ satisfies $|Q_{\mathcal{A}'}| \geq |Q_{\mathcal{A}}|$. Obviously, minimal FSMs may contain neither unreachable nor distinct yet equivalent states, as these could be removed or merged without changing the output function. The question of whether minimal FSMs are unique naturally arises. However, this requires us to first establish a coarser notion of "equality" among FSMs, as an (arbitrary) renaming of the states is sufficient to obtain an FSM that is not *identical* to the original one.

Definition 2.10 (*Isomorphism*)

Let $\mathcal{A}, \mathcal{A}'$ be FSMs of the same type over Σ. An *isomorphism* is a function $f : Q_{\mathcal{A}} \to Q_{\mathcal{A}'}$, satisfying the following conditions:

- f is a bijection, i.e., it is both injective and surjective (note that this requires $|Q_{\mathcal{A}}| = |Q_{\mathcal{A}'}|$),
- $f(q_{0,\mathcal{A}}) = q_{0,\mathcal{A}'}$,
- $\forall q \in Q_{\mathcal{A}}, a \in \Sigma : f(\delta_{\mathcal{A}}(q,a)) = \delta_{\mathcal{A}'}(f(q),a)$, and
- for all $q \in Q_{\mathcal{A}}$, q and $f(q)$ are *locally equivalent*.

In the above definition, we relied on the concept of states being "locally equivalent". This means that, considering the respective FSM type, the states are "equivalent" when considered in isolation, i.e., not taking the whole transition structure into account. For example, in the case of DFAs $\mathcal{A}$ and $\mathcal{A}'$, states $q \in Q_{\mathcal{A}}, q \in Q_{\mathcal{A}'}$ are *locally equivalent* if and only if $q \in F_{\mathcal{A}} \Leftrightarrow q' \in F_{\mathcal{A}'}$. For Mealy machines $\mathcal{M}$ and $\mathcal{M}'$, the corresponding property is $\forall a \in \Sigma : \gamma_{\mathcal{M}}(q,a) = \gamma_{\mathcal{M}'}(q',a)$.

Definition 2.10 allows us to establish a notion of equality that abstracts from a possible renaming or reordering on the states of an FSM.

Definition 2.11

Let $\mathcal{A}, \mathcal{A}'$ be FSMs of the same type. $\mathcal{A}$ and $\mathcal{A}'$ are *isomorphic*, denoted by $\mathcal{A} \simeq \mathcal{A}'$, if and only if there exists an isomorphism $f : Q_{\mathcal{A}} \to Q_{\mathcal{A}'}$.

It should be noted that $\mathcal{A} \simeq \mathcal{A}'$ implies $\mathcal{A} \equiv \mathcal{A}'$, but generally not vice versa.

The original motivation for introducing the concept of isomorphisms was to reason about the uniqueness of minimal FSMs. When "uniqueness" is interpreted modulo isomorphisms, one arrives at the definition of *canonicity*.

Definition 2.12 (*Canonicity*)

Let $\mathcal{A}$ be an FSM that is minimal, i.e., no other FSM $\mathcal{A}'$ with less states computes the same output function $\lambda_{\mathcal{A}}$. If every equivalent FSM $\mathcal{A}'$ with the same number of states is isomorphic to $\mathcal{A}$, $\mathcal{A}$ is called *the canonical* FSM for $\lambda_{\mathcal{A}}$.

Without going into further detail at this point, let us remark that both DFAs and Mealy machines admit canonical forms, i.e., for every DFA $\mathcal{A}$ (Mealy machine $\mathcal{M}$), there exists an equivalent DFA $\mathcal{A}'$ (Mealy machine $\mathcal{M}'$) such that $\mathcal{A}'$ ($\mathcal{M}'$) is canonical.

Suffix output function. A final remark concerns the fact that the output value for a composed word $u \cdot v$ can often be decomposed into a part that is associated with the *prefix* u reaching the state $\mathcal{A}[u]$, and the effect (i.e., the value of the state output function for $\mathcal{A}[u]$) that the *suffix* v has when executed from $\mathcal{A}[u]$ on.

Definition 2.13 (*Suffix output function*)

Let $\mathcal{A}$ be a deterministic FSM over Σ with output domain $\mathcal{D}$. The *suffix output function* of $\mathcal{A}$, $\lambda_{\mathcal{A}}^{suff}$, is defined as

$$\lambda_{\mathcal{A}}^{suff} : \Sigma^* \times \Sigma^* \to \mathcal{D}, \quad \lambda_{\mathcal{A}}^{suff}(u, v) =_{df} \lambda_{\mathcal{A}}^{\mathcal{A}[u]}(v) \;\; \forall u, v \in \Sigma^*.$$

It is typically possible to derive $\lambda_{\mathcal{A}}^{suff}(u, v)$ from $\lambda_{\mathcal{A}}(u \cdot v)$ in a relatively simple way. For example, if $\mathcal{A}$ is a DFA, we have $\lambda_{\mathcal{A}}^{suff}(u, v) = \lambda_{\mathcal{A}}(u \cdot v)$. In the case of a Mealy machine $\mathcal{M}$, $\lambda_{\mathcal{M}}^{suff}(u, v)$ can be obtained from $\lambda_{\mathcal{M}}(u \cdot v)$ by discarding all but the last $|v|$ symbols. Berg *et al.* [31] call an output function with this property *suffix-observable*. In this thesis, we will exclusively consider FSM models with suffix-observable output functions. By slight abuse of notation, we will thus identify the normal and the suffix output function of an FSM $\mathcal{A}$, that is, we write $\lambda_{\mathcal{A}}(u, v)$ instead of $\lambda_{\mathcal{A}}^{suff}(u, v)$, as the presence of a second argument distinguishes it from the normal output function.

3. An Abstract Framework for Active Automata Learning

Active automata learning is the inference of finite-state machines (often DFAs) through *experimentation* (or *testing*). That is, given an output function $\lambda : \Sigma^* \to \mathbb{B}$, the goal is to find a DFA which computes λ merely from the observed values of λ for a certain set of arguments. The term *active* refers to the fact that the *learner* may choose these arguments. However, the learner may only perform one such evaluation of λ ("query") at a time, and the requirement of terminating eventually constrains her to a finite number of queries.

In this chapter, we will develop a mathematical framework allowing us analyze and reason about algorithms with the aim of accomplishing the above task. We start by highlighting important properties of and theorems about regular languages in the next section, which are essential for gaining an understanding about why regular languages are *learnable* in the first place. Afterwards, we will formalize the problem of black-box inference, and establish the frame conditions under which the problem can be tackled by inference algorithms, before describing—on an abstract level—a possible approach that is followed by most existing algorithms.

3.1. Regular Languages, DFAs, and the Myhill-Nerode Theorem

In his seminal work *Three models for the description of language*, Chomsky [51] established a hierarchy of formal languages (i.e., languages over a given finite alphabet Σ, generated by a formal description such as a (formal) grammar) consisting of four classes. While the largest of these classes, $\mathscr{L}_0$ or "type-0 languages", imposes almost no restrictions on the formal description of the languages it contains, each of the other classes—$\mathscr{L}_1$ (type-1) to $\mathscr{L}_3$ (type-3)—imposes additional restrictions, such that each of these three classes is a proper subclass of its preceding classes.

The most restricted class, $\mathscr{L}_3$, is also referred to as the class of *regular languages*. Despite its many restrictions, it constitutes perhaps the most important class of formal languages in theoretical computer science due to its "well-behavedness": the class of regular languages is *closed* under almost any operation, such as concatenation, complementation, or the *Kleene star*, and—given a suitable representation—most of these operations can be computed efficiently.

The mentioned "suitable representation" is that of *deterministic finite automata* (DFAs). It is well-known that the class of regular languages coincides with the class of languages that can be recognized by a DFA. Representing a regular language L in terms of a DFA allows deciding the *membership problem* ("given $w \in \Sigma^*$, is $w \in L$?") in linear time (in the length of w), and admits polynomial-time (in the size of the representing DFAs) algorithms for computing the complement, union, and intersection of regular languages (with the resulting regular language again being represented as a DFA). Furthermore, emptiness ("is $L = \emptyset$?") and universality ("is $L = \Sigma^*$?") can be decided efficiently as well.

An important property of DFAs is that they admit a *canonical minimal form*: for each DFA, there exists an equivalent DFA (i.e., accepting the same regular language) with a minimal number of states. Moreover, this DFA is unique up to isomorphism, and hence called the *canonical* DFA for a regular language (cf. also Section 2.2.5). Again, the canonical DFA can be computed efficiently [88].

It should be noted at this point that some operations, such as concatenating two regular languages, might result in a DFA of exponential size [174], and thus cannot be computed efficiently when restricted to the DFA modeling formalism. They can, however, be computed efficiently in a relaxed modeling formalism, namely that of *non-deterministic finite automata* (NFAs, see Section 2.2.3 for a formal definition). Dropping the requirement of determinism does not change the expressive power (i.e., the class of languages accepted by an NFA is still the class of regular languages), and moreover it still allows the membership problem as well as some properties (such as emptiness) to be decided efficiently. However, deciding other properties such as universality becomes NP-hard. Additionally, the NFA formalism does not admit a minimal canonical form, meaning there may exist several non-isomorphic NFAs with the same minimal size and accepting the same language.

3.1.1. Quotient and DFA Minimization

It has been remarked in the beginning of this section that for each DFA $\mathcal{A}$, there exists an equivalent DFA $\widehat{\mathcal{A}}$ such that $\widehat{\mathcal{A}}$ has a minimal number of states, and that every other DFA $\mathcal{A}'$ that is both equivalent to $\mathcal{A}$ and has the same number of states as $\widehat{\mathcal{A}}$ is isomorphic to $\widehat{\mathcal{A}}$. $\widehat{\mathcal{A}}$ is therefore also called the *canonical* DFA for the output function $\lambda_{\mathcal{A}}$.

Given a DFA $\mathcal{A}$, the canonical DFA $\widehat{\mathcal{A}}$ for $\lambda_{\mathcal{A}}$ can be computed in time $O\big(|Q_{\mathcal{A}}||\Sigma|\log_2|Q_{\mathcal{A}}|\big)$, as has been shown by Hopcroft [88]. Minimization usually consists of two stages: removing *unreachable* states (if $\mathcal{A}$ is not trim), and merging *equivalent* states. The first stage, removing unreachable states, is pretty straightforward: all states in $Q_{\mathcal{A}}$, which cannot be reached from the initial state (i.e., all states in $Q_{\mathcal{A}} \setminus \mathcal{A}[\Sigma^*]$) are removed. The second phase is more involved. The notion of equivalent states has already been introduced in Section 2.2.5: two states q, q' are equivalent ($q \equiv_{\mathcal{A}} q'$) if and only if their state output functions are identical, i.e., $\lambda^q_{\mathcal{A}} = \lambda^{q'}_{\mathcal{A}}$.

Intuitively, merging equivalent states can be done by calculating the equivalence classes of $\equiv_{\mathcal{A}}$, and keeping only one representative of each class (and rerouting transitions to other states in the class to this representative). The formal equivalent of this merging operation is the *quotient* on DFAs.

Definition 3.1 (*DFA Quotient*)

Let $\mathcal{A}$ be a DFA over Σ, and let $\approx \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$ be an equivalence relation over $Q_{\mathcal{A}}$, satisfying the following two conditions:

(i) $\approx$ saturates $F_{\mathcal{A}}$, and

(ii) $\forall q, q' \in Q_{\mathcal{A}} : q \approx q' \Longrightarrow \big(\forall a \in \Sigma : \delta_{\mathcal{A}}(q,a) \approx \delta_{\mathcal{A}}(q',a)\big)$.

The *quotient* DFA $\mathcal{A}/\!\approx = \big\langle Q_{\mathcal{A}/\approx}, \Sigma, q_{0,\mathcal{A}/\approx}, \delta_{\mathcal{A}/\approx}, F_{\mathcal{A}/\approx}\big\rangle$ is then defined as follows:

- $Q_{\mathcal{A}/\approx} =_{df} Q_{\mathcal{A}}/\!\approx$,
- $q_{0,\mathcal{A}/\approx} =_{df} [q_{0,\mathcal{A}}]_{\approx}$,

- $\delta_{\mathcal{A}/\approx}([q]_\approx, a) =_{df} [\delta_{\mathcal{A}}(q,a)]_\approx \quad \forall q \in Q_{\mathcal{A}}, a \in \Sigma$, and
- $F_{\mathcal{A}/\approx} =_{df} \{[q]_\approx \mid q \in F_{\mathcal{A}}\}$.

The conditions that $\approx$ needs to satisfy guarantee that both $\delta_{\mathcal{A}/\approx}$ and $F_{\mathcal{A}/\approx}$ are well-defined, i.e., their definition does not depend on the choice of the representative element $q \in Q_{\mathcal{A}}$ (or $q \in F_{\mathcal{A}}$).

Given a DFA $\mathcal{A}$, the quotient operation allows us to concisely specify the minimal DFA equivalent to $\mathcal{A}$.

Lemma 3.1

Let $\mathcal{A}$ be a DFA *over Σ, let $\mathcal{A}'$ be the trim version of $\mathcal{A}$ (i.e., $Q_{\mathcal{A}'} =_{df} \mathcal{A}[\Sigma^*]$), and let $\equiv_{\mathcal{A}'} \subseteq Q_{\mathcal{A}'} \times Q_{\mathcal{A}'}$ denote the equivalence on states of $\mathcal{A}'$ as defined in Section 2.2.5. Then, $\mathcal{A}'/\equiv_{\mathcal{A}'}$ is the canonical* DFA *for $\lambda_{\mathcal{A}}$.*

3.1.2. The Nerode Congruence

The previous Lemma 3.1 outlines how, given a DFA $\mathcal{A}$ with output function $\lambda_{\mathcal{A}}$, a canonical DFA $\widehat{\mathcal{A}}$ with the same output function can be constructed, namely by merging equivalent states in $\mathcal{A}$. In this section, we will show how $\widehat{\mathcal{A}}$ can be constructed not from an existing DFA, but simply by exploiting properties of an arbitrary output function $\lambda : \Sigma^* \to \mathbb{B}$ that is the characteristic function of a regular language. In the following, we will refer to such output functions as *regular* output functions.

In analogy to regular languages, one can characterize regular output functions as the class of functions $\lambda : \Sigma^* \to \mathbb{B}$ for which a DFA computing them exists. The famous *Myhill-Nerode theorem*[1] [144] provides an alternative characterization of regular output functions, that does not rely on the notion of a DFA. As a first step, we define the *Nerode congruence* on words.

Definition 3.2 (*Nerode congruence*)

Let $\lambda : \Sigma^* \to \mathbb{B}$ be an arbitrary $\mathbb{B}$-valued output function over Σ. The *Nerode congruence* is the binary relation $\cong_\lambda \subseteq \Sigma^* \times \Sigma^*$, defined by

$$u \cong_\lambda u' \Leftrightarrow_{df} \left(\forall v \in \Sigma^* : \lambda(u,v) = \lambda(u',v)\right) \quad \forall u, u' \in \Sigma^*,$$

where $\lambda(u,v)$ is the value of the *suffix output function*, derived from λ in analogy to Definition 2.13.

One easily sees that the Nerode congruence is an equivalence relation on Σ^* that saturates $\lambda^{-1}(1)$. It is also a *right-congruence*, meaning it satisfies

$$\forall u, u', v \in \Sigma^* : u \cong_\lambda u' \Rightarrow u \cdot v \cong_\lambda u' \cdot v.$$

The Nerode congruence is also called *the* syntactical right-congruence. It can be shown that *any* right-congruence $\approx \subseteq \Sigma^* \times \Sigma^*$ saturating $\lambda^{-1}(1)$ refines $\cong_\lambda$.

The intuition behind Definition 3.2 can perhaps better be explained using the concept of *residual output functions*.

[1] In this thesis, we present a slightly modified form of the description, as we reason about output functions, not languages.

Definition 3.3 (*Residual output function*)

Let $\lambda : \Sigma^* \to \mathbb{B}$ be an arbitrary formal language over Σ, and let $u \in \Sigma^*$ be an arbitrary word. The *residual output function* of u with respect to λ, $u^{-1}\lambda$, is defined as

$$(u^{-1}\lambda)(v) =_{df} \lambda(u,v) \;\; \forall v \in \Sigma^*.$$

Obviously, we have $u \cong_\lambda u'$ if and only if $u^{-1}\lambda = u'^{-1}\lambda$.

Residual output functions are the equivalent of state output functions (cf. Definition 2.6) in the case that a DFA $\mathcal{A}$ for the (regular) language is given. In fact, for any state $q \in Q_\mathcal{A}$ of a DFA $\mathcal{A}$ and any word $u \in \Sigma^*$ such that $\mathcal{A}[u] = q$, we have $\lambda^q_\mathcal{A} = u^{-1}\lambda_\mathcal{A}$. More generally, if we define $\sim_\mathcal{A} \subseteq \Sigma^* \times \Sigma^*$ as the equivalence relation that relates words u, u' reaching the same state in $\mathcal{A}$ (i.e., $u \sim_\mathcal{A} u' \Leftrightarrow_{df} \mathcal{A}[u] = \mathcal{A}[u']$), it can be shown that $\sim_\mathcal{A}$ refines $\cong_{\lambda_\mathcal{A}}$. If $\mathcal{A}$ is furthermore canonical, we have $\sim_\mathcal{A} = \cong_{\lambda_\mathcal{A}}$.

The Nerode congruence $\cong_\lambda$ can thus be regarded as the word-level equivalent of the equivalence relation $\equiv_\mathcal{A} \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$, relating equivalent states of a DFA $\mathcal{A}$. It should be noted, however, that $\cong_\lambda$ can be defined for *arbitrary* output functions, not just regular ones. The famous Myhill-Nerode theorem [144] provides a characterization of regular output functions based on $\cong_\lambda$.

Theorem 3.1 (*Myhill-Nerode characterization of regular output functions*)

Let $\lambda : \Sigma^* \to \mathbb{B}$ be a $\mathbb{B}$-valued output function. λ is regular if and only if the index of the Nerode congruence $\cong_\lambda$ is finite.

Proof: We first show that $\cong_\lambda$ has finitely many equivalence classes if λ is regular. In this case, there exists a DFA $\mathcal{A}$ with $\lambda_\mathcal{A} = \lambda$. We have already stated above that the relation $\sim_\mathcal{A}$, relating words reaching the same state in $\mathcal{A}$, refines $\cong_\lambda$, i.e., has at least as many equivalence classes as $\cong_\lambda$. However, $\sim_\mathcal{A}$ cannot have more than $|Q_\mathcal{A}|$ equivalence classes. Since $Q_\mathcal{A}$ is finite, $\cong_\lambda$ can only have finitely many equivalence classes.

For the opposite direction, we need to show that if $\cong_\lambda$ has finitely many equivalence classes, there exists a DFA $\mathcal{A}$ with $\lambda_\mathcal{A} = \lambda$. The proof for this is constructive: let $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, q_{0,\mathcal{A}}, \delta_\mathcal{A}, F_\mathcal{A} \rangle$ be the DFA defined as follows:

- $Q_\mathcal{A} =_{df} \Sigma^*/\cong_\lambda$,
- $q_{0,\mathcal{A}} =_{df} [\varepsilon]_{\cong_\lambda}$,
- $\delta_\mathcal{A}([w]_{\cong_\lambda}, a) =_{df} [w \cdot a]_{\cong_\lambda}$,
- $F_\mathcal{A} =_{df} \{[w]_{\cong_\lambda} \mid \lambda(w) = 1\}$.

To prove that $\mathcal{A}$ computes λ, observe that the construction of $\mathcal{A}$ admits a very simple inductive proof for the fact that, for all $w \in \Sigma^*$, $\mathcal{A}[w] = [w]_{\cong_\lambda}$: the definitions of $q_{0,\mathcal{A}}$ and $\delta_\mathcal{A}$ can be taken *ad verbatim* to form the base case and inductive step, respectively. Thus, $\lambda_\mathcal{A}(w) = 1 \Leftrightarrow \mathcal{A}[w] = [w]_{\cong_\lambda} \in F_\mathcal{A} \Leftrightarrow \lambda(w) = 1$. ■

It should be noted that the construction in the above proof is very similar to the construction of the minimal DFA in Lemma 3.1 by means of the quotient operation (cf. Definition 3.1). In fact, it can easily be generalized to right-congruences other than the Nerode congruence, yielding a variant of the quotient operation on Σ^* that results in a DFA.

3.2. Approximating Regular Languages by Experimentation

We have concluded the previous section by looking at how the canonical DFA $\mathcal{A}$ for an arbitrary regular output function $\lambda : \Sigma^* \to \mathbb{B}$ can be constructed from certain properties of λ (i.e., the equivalence classes of its corresponding Nerode relation). This requires knowledge of the precise definition of λ, as establishing the Nerode relation (or simply determining that two words $u, u' \in \Sigma^*$ are Nerode-inequivalent) requires us to consider the complete, infinite domain Σ^*.

In this section, we will investigate how an *approximation* of the Nerode congruence (and thus the canonical DFA) can be constructed in a setting where we can only inspect finitely many values of λ (and assuming that the input alphabet Σ is known). For the rest of this chapter, we assume that λ is in fact regular, i.e., there exists an (unknown) DFA $\mathcal{A}$ such that $\lambda = \lambda_{\mathcal{A}}$, which we will also refer to as the *target* DFA. Furthermore, we assume $\mathcal{A}$ to be canonical. This makes it easier to reason about the progress of the approximation, even though properties of $\mathcal{A}$ cannot be exploited.

3.2.1. The MAT Framework

An important conceptual contribution by Angluin [19], besides presenting the first polynomial active automata learning algorithm L*, was to establish the framework that made an efficient algorithm possible in the first place: the *Minimally Adequate Teacher* (MAT) model.

In the beginning of this chapter, we already stated that we would be allowed to inspect (finitely many, which is due to the fact that algorithms need to terminate in finite time) values of the output function λ. In the *active* learning setting, the learning algorithm (also referred to as the *learner*) may choose the argument $w \in \Sigma^*$ of λ, and pose a so-called *membership query* (MQ) for w to the *teacher*, who then replies with $\lambda(w)$.

Angluin [20] has shown that using membership queries alone, it is generally not possible to infer the correct target DFA using a polynomial number of membership queries. Moreover, it is easy to see that from the learner's perspective, there is no reasonable stopping criterion without any further input by the teacher: for every finite *sample set* $S \subset \Sigma^*$ for which membership queries have been posed, there are infinitely many different explanations, i.e., distinct, non-isomorphic DFAs whose output on S is consistent with the observations.[2] Further queries may decrease the uncertainty as to whether the conjectured hypothesis is correct (or refute it), but may never eliminate it.

For this reason, Angluin [19] postulated that a teacher, in order to be "minimally adequate", needs to answer a second kind of queries as well: an *equivalence query* (EQ) is posed by the learner for a conjectured DFA $\mathcal{H}$ (the "hypothesis"), and is met with a response from the teacher that either indicates success (i.e., $\lambda = \lambda_{\mathcal{H}}$), or provides a *counterexample*. A counterexample is a word $w \in \Sigma^*$ satisfying $\lambda(w) \neq \lambda_{\mathcal{H}}(w)$, i.e., exposing the inadequacy of the conjectured DFA.[3]

[2] The aforementioned only holds if the number of states of the target DFA is unknown to the learner, but even if it is, an exponential number of membership queries is required [20].

[3] Equivalence queries that provide a counterexample are sometimes referred to (e.g., by de la Higuera [61]) as *strong* equivalence queries, in contrast to *weak* equivalence queries that merely indicate whether the conjectured hypothesis is equivalent to the target DFA or not. Throughout this entire thesis, the term "equivalence query" will always refer to the strong variant, and weak equivalence queries will not be considered at all.

Algorithm 3.1 The "learning loop"

Require: Access to a MAT answering membership and equivalence queries (MQ and EQ) wrt. a target DFA $\mathcal{A}$

Ensure: Hypothesis $\mathcal{H}$ satisfying $\mathcal{H} \equiv \mathcal{A}$

1: Build initial hypothesis $\mathcal{H}$ using MQs
2: **while** EQ($\mathcal{H}$) does not indicate success **do**
3: Let $w \in \Sigma^*$ be the provided counterexample
4: Refine $\mathcal{H}$ using MQs, taking w into account
5: **end while**
6: **return** final hypothesis $\mathcal{H}$

The *Learning Loop*

The availability of membership and equivalence queries immediately gives rise to a general algorithmic skeleton, that virtually all general-purpose active automata learning algorithms build upon. We will refer to this skeleton as the "learning loop", shown as Algorithm 3.1. After an initial *hypothesis construction* phase using membership queries (line 1), the process alternates between posing equivalence queries (line 2) and *hypothesis refinement*, the latter again using membership queries and the provided counterexample (line 4). Thus, active automata learning can be viewed as a special kind of *counterexample-guided refinement*.

The learning loop only terminates when an equivalence query eventually signals success. For this reason, the *partial correctness* of Algorithm 3.1 is trivial: the fact that the result upon termination is correct is guaranteed by the very nature of equivalence queries. Thus, at least from a theoretical standpoint, devising a learning algorithm (by describing how to realize the hypothesis construction and refinement phases) essentially encompasses two aspects: achieving termination by ensuring that eventually a correct hypothesis is conjectured, and doing so as efficiently as possible (i.e., with a minimum number of membership and equivalence queries).

Complexity Measures

A question that naturally arises is what constitutes the actual input to a learning algorithm, as asymptotic complexity is usually given as a function of the input *size*. Generally, the following three parameters are considered:

- $n =_{df} |\Sigma^*/\cong_\lambda|$, the size of the (canonical) target DFA $\mathcal{A}$,
- $k =_{df} |\Sigma|$, the size of the input alphabet (sometimes also treated as a constant), and
- m, the length of the longest counterexample returned by an equivalence query (note that the generation of counterexamples is *not* under the learner's control, thus they also constitute an input to the algorithm).

While the first two parameters, n and k, are determined entirely by the target DFA $\mathcal{A}$, the last one, m, is determined by the teacher. In most cases, (adversarial) teachers can provide counterexamples of arbitrary length, making it hard for the learner to analyze them. Assuming that a learner never conjectures a hypothesis of size bigger than n, a *cooperative teacher* [175] can

always respond with a counterexample of length $O(n)$. On the other hand, for many combinations of conjectured hypothesis and target DFA, every counterexample is of length $\Omega(n)$. Since in most practical applications the teacher cannot be assumed to be cooperative, $m = \Omega(n)$ usually is a reasonable assumption.

Analyzing the complexity of an active automata learning algorithm differs from the analysis of other algorithms in that the time spent on "raw" computations, such as for manipulating data structures, is usually neglected in favor of the *query complexity*, i.e., the (asymptotic) number of membership and equivalence queries posed by a learner. The reason for this is that the computation time is usually a low-order polynomial in the above parameters, and is in practice always dominated by the time spent on queries. Thus, the following complexity measures are used to assess the performance of a learning algorithm, all of which are usually specified asymptotically and as a function of n, k, and m:

- *(membership) query complexity*, the number of membership queries posed by a learner,
- *equivalence query complexity*, the number of equivalence queries posed by a learner, and
- *symbol complexity*, the overall number of symbols contained in all words for which membership queries have been posed.

Of these measures, the first is usually regarded as the most important one. The equivalence query complexity is usually neglected, as it is easy to establish that no more than $n-1$ equivalence queries need to be made (by ensuring that, starting with a one-state initial hypothesis, every counterexample gives rise to at least one additional state). Furthermore, Balcázar *et al.* [25] have shown that this upper bound cannot be significantly lowered without forsaking a polynomial membership query complexity.

The *symbol complexity* has long been neglected in favor of a uniform cost model for membership queries, regardless of their length. Isberner *et al.* [110] point out that this is not sufficient, as in many practical applications of active automata learning, the cost for realizing a membership query is linear in the length of the respective word. On the other hand, Choi *et al.* [50] describe a scenario with a very high fixed cost per membership query, where the length of the query is indeed mostly negligible. As theoretical considerations should be oblivious of concrete application scenarios, membership query and symbol complexities should be regarded as two independent cost measures.

3.2.2. Black-Box Classification Schemes

After having established the precise frame conditions in which an active automata learning algorithm operates, we can now begin with establishing our abstract framework for learning algorithms, in order to shed further light on how and why the inference of the correct target DFA works.

In the introduction to Section 3.2, we have stated our goal of approximating the Nerode congruence. In the most general sense, we are thus looking for a way to determine the equivalence class of an arbitrary word $u \in \Sigma^*$ wrt. some equivalence relation that approximates the Nerode congruence $\cong_\lambda$.

Definition 3.4 (*Black-box classifier*)

A *black-box classifier* is a function $\kappa : \Sigma^* \to \mathscr{C}$, where $\mathscr{C}$ is an arbitrary *class domain*.

κ is called a valid *over-approximation* (or simply *valid*) wrt. some output function λ if and only if

$$\forall u, u' \in \Sigma^* : \kappa(u) \neq \kappa(u') \Rightarrow u \not\cong_\lambda u'.$$

Without further knowledge about the definition of λ, the only way to establish this is to maintain *witnesses* that prove the inequivalence (wrt. $\cong_\lambda$) of two words $u, u' \in \Sigma^*$ satisfying $\kappa(u) \neq \kappa(u')$. Therefore, in the case that $\kappa(u) \neq \kappa(u')$, the classifier κ needs to establish $\lambda(u, v) \neq \lambda(u', v)$ for at least one $v \in \Sigma^*$.

Definition 3.5 (*Suffix-based black-box classifier*)

Let Σ be an arbitrary input alphabet. A (finite) *suffix-based black-box classifier* is a black-box classifier $\kappa : \Sigma^* \to \mathscr{C}$, where $\mathscr{C} = \{f : \Sigma^* \rightharpoonup \mathbb{B} \mid |\mathrm{dom}\, f| < \infty\}$, and for all $u, u' \in \Sigma^*$ such that $\kappa(u) \neq \kappa(u')$, we have

$$\exists v \in \mathrm{dom}\,\kappa(u) \cap \mathrm{dom}\,\kappa(u') : \kappa(u)(v) \neq \kappa(u')(v).$$

κ is called *valid* for some suffix-observable output function $\lambda : \Sigma^* \to \mathbb{B}$ if and only if

$$\forall u \in \Sigma^* : \forall v \in \mathrm{dom}\,\kappa(u) : \kappa(u)(v) = \lambda(u, v).$$

The set of all valid suffix-based black-box classifiers for λ is denoted by K_λ.

It is easy to see that a suffix-based black-box classifier $\kappa \in K_\lambda$ that is valid for λ in the sense of the above definition is also valid in the sense of Definition 3.4 due to the existence of a *separating suffix* (or *separator*) v. Note that this implies that any two $f, f' \in \kappa(\Sigma^*)$ either have a non-empty intersection of their domains, or $\kappa(\Sigma^*)$ is a singleton containing only the function that is nowhere defined.

In the following, we will only consider suffix-based black-box classifiers, and use the term "black-box classifier" synonymously.

Definition 3.6 (*Characterizing set, separator set*)

Let $\lambda : \Sigma^* \to \mathbb{B}$ be an output function, and let $\kappa \in K_\lambda$ be a black-box classifier for λ.

- The *characterizing set* (wrt. κ) of a word $u \in \Sigma^*$, $Ch_\kappa(u)$, is defined as

$$Ch_\kappa(u) =_{df} \mathrm{dom}\,\kappa(u).$$

- The *separator set* (wrt. κ) of words $u, u' \in \Sigma^*$, $Seps_\kappa(u, u')$, is defined as

$$Seps_\kappa(u, u') =_{df} \{v \in Ch_\kappa(u) \cap Ch_\kappa(u') \mid \kappa(u)(v) \neq \kappa(u')(v)\}.$$

The characterizing set for a word (also called *prefix*, as it is combined with a *suffix* when evaluating λ) $u \in \Sigma^*$ can be regarded as the set of *suffixes* $v \in \Sigma^*$ that are being tested (by evaluating $\lambda(u, v)$) to determine the equivalence class of u. Since the point of the suffixes in the characterizing set is to discriminate between equivalence classes, they are also referred to as *discriminators*. As has been remarked above, the characterizing set of two prefixes must always intersect,

or be empty for all prefixes. The separator set, on the other hand, contains the evidence for two prefixes $u, u' \in \Sigma^*$ being inequivalent under κ, i.e., $Seps_\kappa(u,u') = \emptyset$ if and only if $u \sim_\kappa u'$. By slight abuse of notation, we will also sometimes use equivalence classes of $\sim_\kappa$ as arguments of $Seps_\kappa$, i.e., $Seps_\kappa([u]_\kappa,[u']_\kappa) = Seps_\kappa(u,u')$, which is apparently well-defined.

Remark 3.1

It makes sense to establish that the relation $\sim_\kappa$, besides being refined by $\cong_\lambda$, furthermore saturates $\lambda^{-1}(1)$. This can be ensured by enforcing $\varepsilon \in Ch_\kappa(u)$ for all $u \in \Sigma^*$. This property greatly simplifies many proofs in this chapter, which is why we will implicitly assume that it holds for all black-box classifiers that we consider.

Probably the simplest way of defining a valid black-box classifier is to introduce a *global* set of suffixes $\mathcal{V} \subset \Sigma^*$ that homogeneously applies to all prefixes.

Definition 3.7 (*Global suffix-based classifier*)

Let $\kappa : \Sigma^* \to \mathscr{C}$ be a suffix-based classifier. κ is called *global* if for all $u, u' \in \Sigma^*$, we have

$$Ch_\kappa(u) = Ch_\kappa(u') = \mathcal{V}$$

for some *global suffix set* $\mathcal{V} \subset \Sigma^*$.

A prominent example of a global suffix-based classifier is the *observation table*, which forms the main data structure of the L* algorithm [19] and some of its derivatives. We will elaborate on this in Section 3.4.1.

Black-Box Abstractions

So far we have only considered how a classification of words, that is guaranteed to be refined by the Nerode congruence $\cong_\lambda$, can be established via a black-box classifier κ. However, identifying the corresponding equivalence classes in $\Sigma^*/\sim_\kappa$ still requires to potentially consider the entire infinite domain Σ^*. Thus, we need to maintain information about these equivalence classes (or rather their representatives, as the classes might be infinite) as well.

Definition 3.8 (*Black-box abstraction*)

Let $\lambda : \Sigma \to \mathbb{B}$ be an arbitrary output function. A *black-box abstraction* of λ is a pair $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$, where $\mathcal{U} \subset \Sigma^*$ is a finite set of *short prefixes* satisfying $\varepsilon \in \mathcal{U}$, and $\kappa \in K_\lambda$ is a black-box classifier for λ. The *set of classes* of $\mathcal{R}$, $\mathscr{C}(\mathcal{R})$, is defined as $\mathscr{C}(\mathcal{R}) =_{df} \{[u]_\kappa \mid u \in \mathcal{U}\}$.

We have remarked in Section 3.1.2 that any relation that is strictly refined by the Nerode congruence while saturating $\lambda^{-1}(1)$ cannot be a right-congruence. Thus, applying a quotient construction as in the proof of Theorem 3.1 in general is not well-defined. However, limiting the definition to the representatives in $\mathcal{U}$, a weaker requirement suffices.

Definition 3.9 (*Closedness, determinism*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction. $\mathcal{R}$ is called ...

(i) *closed* if and only if for all short prefixes $u \in \mathcal{U}$ and all symbols $a \in \Sigma$, there exists a short prefix $u' \in \mathcal{U}$ such that $ua \sim_\kappa u'$ (i.e., $[ua]_\kappa \in \mathscr{C}(\mathcal{R})$).

(ii) *deterministic*[4] if and only if for all short prefixes $u, u' \in \mathcal{U}$ and all symbols $a \in \Sigma$, we have

$$u \sim_\kappa u' \Rightarrow ua \sim_\kappa u'a.$$

The following definition details the quotient construction for closed and deterministic black-box abstractions.

Definition 3.10

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ a closed and deterministic black-box abstraction. The DFA corresponding to $\mathcal{R}$, DFA($\mathcal{R}$), is the DFA $\mathcal{H}$, where

- $Q_\mathcal{H} =_{df} \{[u]_\kappa \mid u \in \mathcal{U}\}$,
- $q_{0,\mathcal{H}} =_{df} [\varepsilon]_\kappa$,
- $\delta_\mathcal{H}([u]_\kappa, a) =_{df} [ua]_\kappa \quad \forall u \in \mathcal{U}, a \in \Sigma$, and
- $F_\mathcal{H} =_{df} \{[u]_\kappa \mid u \in \mathcal{U}, \kappa(u)(\varepsilon) = 1\}$.

Remark 3.2

It is crucial to observe that, for a black-box abstraction $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$, states of $\mathcal{H} = \text{DFA}(\mathcal{R})$ are identified with equivalence classes of $\sim_\kappa$ (more precisely: those equivalence classes that have an element in $\mathcal{U}$). This identification will be exploited in the following, as it allows us to reason about statements such as "$\mathcal{H}[u] = [u]_\kappa$", i.e., does the equivalence class of the state reached by u in $\mathcal{H}$ match the equivalence class of u itself?

As equivalence classes of $\sim_\kappa$ are subsets of Σ^*, we can refer to the *representative elements* of some equivalence class $[u]_\kappa$ ($u \in \Sigma^*$) via $[u]_\kappa \cap \mathcal{U}$. This notation however easily leads to confusion. Therefore, we will always refer to the representatives by means of a special mapping $\rho_\mathcal{R} : \Sigma^*/\sim_\kappa \to 2^\mathcal{U}$, defined via $\rho_\mathcal{R}([u]_\kappa) =_{df} [u]_\kappa \cap \mathcal{U}$ for all $u \in \Sigma^*$.

Consistency Properties

The question of how well the DFA $\mathcal{H} = \text{DFA}(\mathcal{R})$ reflects the information contained in $\kappa(u)$, $u \in \mathcal{U}$, naturally arises. It would be desirable if, for all $u \in \mathcal{U}$ and $v \in Ch_\kappa(u)$, we had $\lambda_\mathcal{H}(u, v) = \kappa(u)(v) = \lambda(u, v)$. However, we will see that there are multiple reasons for why this might not be the case.

Definition 3.11 (*Reachability inconsistency, reachability (in)consistent*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction, and let $\mathcal{H} = \text{DFA}(\mathcal{R})$ be the corresponding DFA. A word $u \in \Sigma^*$ constitutes a *reachability inconsistency* (wrt. $\mathcal{R}$) if and only if $\mathcal{H}[u] \neq [u]_\kappa$.

$\mathcal{R}$ is called *reachability inconsistent* if and only if there exists $u \in \mathcal{U}$ such that u constitutes a reachability inconsistency. Otherwise, i.e., if $\forall u \in \mathcal{U} : [u]_\kappa = \mathcal{H}[u]$, $\mathcal{R}$ is called *reachability consistent*.

[4] Angluin [19] calls this property "consistency". It is however more adequate to view it as a case of apparent nondeterminism, caused by an overly coarse abstraction. The term "consistency", furthermore, is more adequate for two properties that we will define in the next section.

The obvious reason for possible violations of reachability consistency is the fact that $\sim_\kappa$ cannot be a right-congruence (unless $\sim_\kappa = \cong_\lambda$), hence the (inductive) correctness proof of Theorem 3.1 is not applicable. However, it is possible to ensure that $\sim_\kappa$ behaves like a right-congruence when restricted to $\mathcal{U}$ (and thus ensuring reachability consistency of $\mathcal{R}$) by enforcing a simple syntactic property.

Lemma 3.2

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction. If $\mathcal{U}$ is prefix-closed, $\mathcal{R}$ is reachability consistent.

Proof: Assume that $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ is a closed and deterministic black-box abstraction such that $\mathcal{U}$ is prefix-closed, but $\mathcal{R}$ is reachability inconsistent, i.e., there exists $u \in \mathcal{U}$ such that $\mathcal{H}[u] \neq [u]_\kappa$. Assume w.l.o.g. that u is a shortest such element of $\mathcal{U}$. Since $\mathcal{H}[\varepsilon] = q_{0,\mathcal{H}} = [\varepsilon]_\kappa$ by definition, we can infer that $|u| \geq 1$. Thus, we can decompose u into $u = u' \cdot a$, where $u' \in \mathcal{U}$ (due to prefix-closedness) and $a \in \Sigma$. As we have assumed u to be a shortest violating prefix, we have $\mathcal{H}[u'] = [u']_\kappa$. However, from the definition of $\mathcal{H}[\cdot]$, we know that $\mathcal{H}[u] = \delta_\mathcal{H}(\mathcal{H}[u'], a)$, which according to Definition 3.10 is $[u'a]_\kappa = [u]_\kappa$, contradicting our assumption that u is a prefix violating reachability consistency. ■

Even with reachability consistency established, it is not guaranteed that $\mathcal{H}$ correctly reflects the observed behavior of λ. We start by formally defining the (in-)consistency property.

Definition 3.12 (*Output inconsistency, output (in-)consistent*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction, and let $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$ be the corresponding DFA. A pair $(u, v) \in \mathcal{U} \times \Sigma^*$ constitutes an *output inconsistency* (wrt. $\mathcal{R}$) if and only if $\lambda_\mathcal{H}^{[u]_\kappa}(v) \neq \lambda(u, v)$.

$\mathcal{R}$ is called *output inconsistent* if and only if there exist $u \in \mathcal{U}, v \in Ch_\kappa(u)$ such that (u, v) constitutes an output inconsistency. Otherwise, i.e., if

$$\forall u \in \mathcal{U} : \forall v \in Ch_\kappa(u) : \lambda_\mathcal{H}^{[u]_\kappa}(v) = \lambda(u, v) = \kappa(u)(v),$$

$\mathcal{R}$ is called *output consistent*.

Violations of output consistency of a black-box abstraction $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ are caused by the fact that the construction of $\mathcal{H}$ in Definition 3.10 does not establish any connection between the value of $\kappa(u)(v)$, $u \in \mathcal{U}, v \in Ch_\kappa(u)$, and whether the state $\delta_\mathcal{H}([u]_\kappa, v)$ is accepting (except for $v = \varepsilon$). Thus, the information about the output behavior for longer suffixes v is not propagated when the transition structure of $\mathcal{H}$ is being constructed.

Steffen *et al.* [167] introduced the concept of *semantic suffix closedness* to maintain output consistency. However, Van Heerdt [86] has shown that their definition is insufficient, i.e., does not ensure output consistency in the above sense. Furthermore, the definition is specific to the used data structure. We thus give an improved and generalized definition of this concept, and show that it indeed ensures output consistency.

Definition 3.13 (*Semantic suffix-closedness*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction. $\mathcal{R}$ is called *semantically suffix-closed* if and only if for all prefixes $u \in \mathcal{U}$ and all suffixes $v \in Ch_\kappa(u)$ such that $v = a \cdot v'$, we have $v' \in Ch_\kappa(u \cdot a)$.

Lemma 3.3

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction. If $\mathcal{R}$ is semantically suffix-closed, then $\mathcal{R}$ is output consistent.

Proof: Assume that $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ is a closed, deterministic, and semantically suffix-closed black-box abstraction, that however is not output consistent. Therefore, there exists a prefix $u \in \mathcal{U}$ and suffix $v \in Ch_\kappa(u)$ such that $\lambda_{\mathcal{H}}^{[u]_\kappa}(v) \neq \kappa(u)(v)$, where $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$. We furthermore assume that u and v are chosen such that v is a shortest (over all possible choices for u) violating suffix.

The definition of $F_{\mathcal{H}}$ in Definition 3.10 guarantees that $\lambda_{\mathcal{H}}^{[u]_\kappa}(\varepsilon) = \kappa(u)(\varepsilon)$. In particular, this implies that the above violating suffix v cannot be empty. Thus, we can decompose it into $v = a \cdot v'$, where $v' \in Ch_\kappa(ua)$ due to semantic suffix-closedness. As v was chosen to be the shortest violating suffix, we can infer that, for all $u' \in \mathcal{U}$ such that $u' \sim_\kappa ua$, $\lambda_{\mathcal{H}}^{[u']_\kappa}(v') = \kappa(u')(v') = \kappa(ua)(v') = \kappa(u)(v)$. Hence, $\lambda_{\mathcal{H}}^{[u]_\kappa}(v) = \lambda_{\mathcal{H}}^{[u']_\kappa}(v') = \kappa(u')(v') = \kappa(u)(v)$, contradicting our assumption that (u, v) constituted an output inconsistency. ■

A relatively easy way to establish semantic suffix-closedness is to use a global suffix-based abstraction, and ensure that the global suffix set $\mathcal{V}$ remains suffix-closed (the L* algorithm follows this approach). Maintaining semantic suffix-closedness in other settings requires significantly more work, as Chapter 5 of this thesis will show.

We conclude our description of consistency properties with the following statement.

Corollary 3.1 (*Observation consistency*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be black-box abstraction of some output function $\lambda : \Sigma^* \to \mathbb{B}$. If $\mathcal{R}$ is both reachability and output consistent, we have

$$\forall u \in \mathcal{U} : \forall v \in Ch_\kappa(u) : \lambda_{\mathcal{H}}(u, v) = \lambda(u, v).$$

Proof: Let $u \in \mathcal{U}$ and $v \in Ch_\kappa(u)$ be chosen arbitrarily. By definition, we have $\lambda_{\mathcal{H}}(u, v) = \lambda_{\mathcal{H}}^{\mathcal{H}[u]}(v)$. Reachability consistency guarantees $\mathcal{H}[u] = [u]_\kappa$, and output consistency ensures $\lambda_{\mathcal{H}}^{[u]_\kappa}(v) = \kappa(u)(v) = \lambda(u, v)$. ■

Correctness and Termination

Reachability and output consistency ensure that the *behavior* observed when evaluating λ during the classification using κ is correctly reflected in the constructed DFA $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$. The following lemma, which is a generalized version of the one given by Isberner and Steffen [108], states the guarantees that can be made about the *structural* relation between $\mathcal{H}$ and the unknown target DFA $\mathcal{A}$.

Lemma 3.4 (*Invariants of black-box abstractions*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction of a regular output function $\lambda = \lambda_{\mathcal{A}}$, and let $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$ be the DFA *for $\mathcal{R}$. Then, the following invariants hold:*

(I1) *Inequivalent (with respect to $\sim_\kappa$) prefixes in $\mathcal{U}$ lead to different states in $\mathcal{A}$:*

$$\forall u, u' \in \mathcal{U} : u \not\sim_\kappa u' \Rightarrow \mathcal{A}[u] \neq \mathcal{A}[u'].$$

(I2) *The acceptance of a state in $\mathcal{H}$ corresponding to a prefix in $\mathcal{U}$ is correct:*

$$\forall u \in \mathcal{U} : [u]_\kappa \in F_\mathcal{H} \Leftrightarrow \mathcal{A}[u] \in F_\mathcal{A}.$$

(I3) *If both a state in $\mathcal{A}$ and its a-successor ($a \in \Sigma$) have been discovered by prefixes in $\mathcal{U}$, the corresponding transition in $\mathcal{H}$ is correct:*

$$\forall u, u' \in \mathcal{U}, a \in \Sigma : \mathcal{A}[ua] = \mathcal{A}[u'] \Rightarrow \delta_\mathcal{H}([u]_\kappa, a) = [u']_\kappa.$$

Proof:

- (I1): Let $u, u' \in \mathcal{U}$ be such that $u \not\sim_\kappa u'$. There then exists a suffix $v \in Seps_\kappa(u, u')$ such that $\lambda(u,v) = \kappa(u)(v) \neq \kappa(u')(v) = \lambda(u',v)$. Thus $\lambda_\mathcal{A}^{\mathcal{A}[u]}(v) \neq \lambda_\mathcal{A}^{\mathcal{A}[u']}(v)$, and hence $\mathcal{A}[u] \neq \mathcal{A}[u']$.
- (I2): By Definition 3.10, we have $[u]_\kappa \in F_\mathcal{H} \Leftrightarrow \kappa(u)(\varepsilon) = 1$. Since $\kappa(u)(\varepsilon) = \lambda(u,\varepsilon) = \lambda_\mathcal{A}^{\mathcal{A}[u]}(\varepsilon)$, we can conclude that $[u]_\kappa \in F_\mathcal{H} \Leftrightarrow \mathcal{A}[u] \in F_\mathcal{A}$.
- (I3): Let $u, u' \in \mathcal{U}, a \in \Sigma$ be such that $\mathcal{A}[ua] = \mathcal{A}[u']$. $\delta_\mathcal{H}([u]_\kappa, a) \neq [u']_\kappa$ would imply $ua \not\sim_\kappa u'$, which however cannot be the case, as it—in conjunction with (I1)—would contradict $\mathcal{A}[ua] = \mathcal{A}[u']$. ■

Active automata learning is sometimes also referred to as *regular extrapolation*, resembling polynomial extrapolation: from a finite number of *supports* (i.e., pairs of x and y values of some unknown target function), a polynomial function is inferred that "explains" the given values. Similarly, from a finite number of observations, an automaton is inferred that is consistent with the observations. Note that the extrapolation step in active automata learning is the construction of the DFA: the black-box classifier over-approximates the Nerode congruence, but evaluating it for arbitrary words requires evaluating λ. By extrapolating the transition structure (up to the observable granularity of the classifier) from $\mathcal{U}$ to Σ^*, an additional *extrapolation error* is introduced.

It is a well-known result from polynomial extrapolation that if the target function is a polynomial of degree d, and the number of supports is at least $d+1$, the extrapolated polynomial will be identical to the target function, i.e., the extrapolation error vanishes entirely. A similar result exists for the case of regular extrapolation.

Theorem 3.2 (*Zero-error theorem of black-box abstractions*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction of an output function $\lambda = \lambda_\mathcal{A}$, where $\mathcal{A}$ is the canonical DFA *for λ. If $|\mathscr{C}(\mathcal{R})| = |\Sigma^*/\cong_\lambda|$, then …*

(i) $\mathcal{R}$ is necessarily closed and deterministic, thus $\mathrm{DFA}(\mathcal{R})_{df} = \mathcal{H}$ *is defined,*

(ii) $\lambda_\mathcal{H} = \lambda$ (in particular, $\mathcal{R}$ is reachability and output consistent), and

(iii) $\mathcal{H}$ *is isomorphic to* $\mathcal{A}$.

Proof:

(i) Assume that $\mathcal{R}$ is not closed, i.e., there exists $u \in \mathcal{U}$ and $a \in \Sigma$ such that for all $u' \in \mathcal{U}$, $ua \not\sim_\kappa u'$. Then, $|\Sigma^*/\sim_\kappa| > |\mathscr{C}(\mathcal{R})| = |\Sigma^*/\cong_\lambda|$, contradicting the validity requirement that $\cong_\lambda$ refines $\sim_\kappa$.

If $\mathcal{R}$ is not deterministic, there exists $u, u' \in \mathcal{U}$ and $a \in \Sigma$ such that $u \sim_\kappa u'$, but $ua \not\sim_\kappa u'a$. Let $v \in Seps_\kappa(ua, u'a)$ be a separator for ua and $u'a$, i.e., $\kappa(ua)(v) = \lambda(ua, v) \neq \lambda(u'a, v) = \kappa(u'a, v)$, thus $a \cdot v$ proves that $u \not\cong_\lambda u'$. Since $\sim_\kappa$ refines $\cong_\lambda$, we have $[u]_{\cong_\lambda}, [u']_{\cong_\lambda} \subseteq [u]_\kappa$ and, since $u \not\cong_\lambda u'$, $[u]_{\cong_\lambda} \cap [u']_{\cong_\lambda} = \emptyset$. Thus, $[u]_\kappa$ is the union of at least two distinct equivalence classes of $\cong_\lambda$, which implies $|\Sigma^*/\cong_\lambda| > |\Sigma^*/\sim_\kappa|$, contradicting the assumption.

(ii) Follows directly from (iii).

(iii) Let $\mathcal{U}' \subseteq \mathcal{U}$ be a subset of *representatives* of $\mathcal{U}$ such that $\varepsilon \in \mathcal{U}'$, and for every element $u \in \mathcal{U}$ there exists exactly one $u' \in \mathcal{U}'$ such that $u \sim_\kappa u'$ (i.e., all elements of $\mathcal{U}'$ are pairwisely inequivalent wrt. $\sim_\kappa$). Apparently, $\{[u]_\kappa \mid u \in \mathcal{U}'\} = \mathscr{C}(\mathcal{R})$. Let furthermore $f : Q_\mathcal{H} \to Q_\mathcal{A}$ be a function mapping states of $\mathcal{H}$ (i.e., elements of $\mathscr{C}(\mathcal{R})$) to states of $\mathcal{A}$, defined by $f([u]_\kappa) =_{df} \mathcal{A}[u]$ for all $u \in \mathcal{U}'$. Applying the invariants introduced in Lemma 3.4, we now show that f is an isomorphism. First, observe that f is injective due to (I1), and since $|Q_\mathcal{H}| = |\mathscr{C}(\mathcal{R})| = |\Sigma^*/\cong_\lambda| = |Q_\mathcal{A}|$, f is a bijection.

- $f(q_{0,\mathcal{H}}) = f([\varepsilon]_\kappa) = \mathcal{A}[\varepsilon] = q_{0,\mathcal{A}}$ by definition.
- Let $u \in \mathcal{U}', a \in \Sigma$ be chosen arbitrarily, and let $u' \in \mathcal{U}'$ be such that $\mathcal{A}[ua] = \mathcal{A}[u']$ (note that such a u' must exist, as $\mathcal{A}[ua]$ has a preimage under f). Applying (I3) yields $\delta_\mathcal{H}([u]_\kappa, a) = [u']_\kappa$. Thus, $f(\delta_\mathcal{H}([u]_\kappa, a)) = f([u']_\kappa) = \mathcal{A}[u'] = \mathcal{A}[ua] = \delta_\mathcal{A}(\mathcal{A}[u], a) = \delta_\mathcal{A}(f([u]_\kappa), a)$.
- $[u]_\kappa \in F_\mathcal{H} \Leftrightarrow \mathcal{A}[u] = f([u]_\kappa) \in F_\mathcal{A}$ follows directly from (I2). ■

3.2.3. Refining Black-Box Abstractions

We have concluded the previous section with Theorem 3.2, stating that the extrapolation error vanishes entirely if a black-box abstraction $\mathcal{R}$ has reached the granularity of the Nerode congruence corresponding to the unknown regular target function λ.

Until now, we have not discussed how, starting with a trivial initial black-box abstraction, this level of granularity can eventually be achieved. Before we continue, it is helpful to first formalize the notion of refinement between black-box abstractions. We start by defining refinement on the level of black-box classifiers, which goes beyond refinement of their equivalence kernels only.

Definition 3.14 (*Refinement of black-box classifiers*)

Let $\kappa, \kappa' \in K_\lambda$ be black-box classifiers for some output function $\lambda : \Sigma^* \to B$. κ' *refines* κ, denoted by $\kappa' \sqsubseteq \kappa$, if and only if:

- $\sim_{\kappa'}$ refines $\sim_\kappa$ (i.e., for all $u, u' \in \Sigma^*$, we have $\kappa'(u) = \kappa'(u') \Rightarrow \kappa(u) = \kappa(u')$), and

- for all $u \in \Sigma^*$, we have $Ch_{\kappa'}(u) \supseteq Ch_{\kappa}(u)$.

The refinement is *strict* (denoted by $\kappa' \sqsubset \kappa$) if and only if $\sim_{\kappa'}$ strictly refines $\sim_{\kappa}$. This implies that there exists $u \in \Sigma^*$ such that $Ch_{\kappa'}(u) \supset Ch_{\kappa}(u)$.

Definition 3.15 (*Refinement of black-box abstractions*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction. A black-box abstraction $\mathcal{R}' = \langle \mathcal{U}', \kappa' \rangle$ is said to *refine* $\mathcal{R}$, denoted by $\mathcal{R}' \sqsubseteq \mathcal{R}$, if and only if:

- $\mathcal{U}' \supseteq \mathcal{U}$, and
- $\kappa' \sqsubseteq \kappa$.

We say that $\mathcal{R}'$ *strictly refines* $\mathcal{R}$ ($\mathcal{R}' \sqsubset \mathcal{R}$) if and only if $\mathcal{R}' \sqsubseteq \mathcal{R}$, and $\mathscr{C}(\mathcal{R}') > \mathscr{C}(\mathcal{R})$.[5]

Establishing the first of the refinement conditions in the above definition, augmenting $\mathcal{U}$, is straightforward. To describe how a (suffix-based) black-box classifier κ can be modified in a way that preserves the restrictions of Definition 3.4 while satisfying those of the above Definition 3.15, we introduce the concept of *splitting* classes of $\sim_{\kappa}$.

Definition 3.16

Let $\kappa \in K_{\lambda}$ be a suffix-based black-box classifier of some output function λ. A *split* of κ with respect to a class $C \subseteq \Sigma^*$ and a suffix (or discriminator) $v \in \Sigma^*$ is defined as follows:

$$\text{split}\colon K_{\lambda} \times 2^{\Sigma^*} \times \Sigma^* \to K_{\lambda}$$

$$\text{split}(\kappa, C, v)(u) =_{df} \begin{cases} \kappa(u) \cup \{v \mapsto \lambda(u,v)\} & \text{if } u \in C \\ \kappa(u) & \text{otherwise} \end{cases}$$

Note that C must be saturated by $\sim_{\kappa}$ in order to ensure that $\text{split}(\kappa, C, v)$ is a valid black-box classifier, i.e., obeys the restrictions of Definitions 3.4 and 3.5.

It is easy to see that, if C is saturated by $\sim_{\kappa}$, $\text{split}(\kappa, C, v)$ refines κ, and furthermore that if there exists $u, u' \in C$ such that $\lambda(u,v) \neq \lambda(u',v)$, $\text{split}(\kappa, C, v)$ strictly refines κ.

Remark 3.3

The above definition of the split function results in the coarsest refinement κ' of κ satisfying $\forall u \in C : v \in Ch_{\kappa'}(u)$. Depending on superimposed syntactical constraints, learning algorithms might use an even more refined classifier in the situations where the split function is used in the following. For example, if the algorithm uses a global suffix-based classifier, the new classifier might simply be obtained by adding v to the global suffix set $\mathcal{V}$, thus satisfying the above property while preserving the global characteristics of the classifier. Similarly, if the suffixes are maintained in a (semantically) suffix-closed fashion, further suffixes might be added to the characterizing sets as well. Since all of these classifiers obeying additional syntactical constraints refine κ', the correctness of the lemmas and theorems in this chapter remains unaffected.

[5] Note that the introduced notation, while intuitive, may lead to unexpected results: there may exist black-box abstractions $\mathcal{R}, \mathcal{R}'$ such that $\mathcal{R}' \sqsubseteq \mathcal{R}$, $\mathcal{R} \not\sqsubseteq \mathcal{R}'$, and yet $\mathcal{R}' \not\sqsubset \mathcal{R}$.

In Section 3.2.1 we have remarked that in the general setting, refinements of the hypothesis (which is induced by a black-box abstraction) are triggered by counterexamples. However, in some cases a black-box abstraction itself contains enough information to derive a refined version of it. This is the case if it is impossible to construct a DFA from it, or if one of the consistency properties (Definitions 3.11 and 3.12) are violated.

Lemma 3.5

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction of an output function λ. If $\mathcal{R}$ is not closed or not deterministic, then there exists a black-box abstraction $\mathcal{R}' = \langle \mathcal{U}', \kappa' \rangle$ such that $\mathcal{R}' \sqsubset \mathcal{R}$.

Proof: First, assume that $\mathcal{R}$ is not closed. There then exists $u \in \mathcal{U}$ and $a \in \Sigma$ such that for all $u' \in \mathcal{U}$, $ua \not\sim_\kappa u'$. Hence, by choosing $\mathcal{U}' =_{df} \mathcal{U} \cup \{ua\}$, we establish $|\mathscr{C}(\mathcal{R}')| > |\mathscr{C}(\mathcal{R})|$ (as $\kappa' = \kappa$, thus $\mathscr{C}(\mathcal{R}) \subseteq \mathscr{C}(\mathcal{R}')$ and $[ua]_{\sim_\kappa} \in \mathscr{C}(\mathcal{R}') \setminus \mathscr{C}(\mathcal{R})$) and therefore $\mathcal{R}' \sqsubset \mathcal{R}$.

Let us now consider the case that $\mathcal{R}$ is not deterministic. Then, there exist $u, u' \in \mathcal{U}$ and $a \in \Sigma$ such that $u \sim_\kappa u'$, but $ua \not\sim_\kappa u'a$. Let $v \in Seps_\kappa(ua, u'a)$ be a separator, i.e., $\kappa(ua)(v) \neq \kappa(u'a)(v)$. Consider the black-box classifier κ' obtained from κ by splitting the equivalence class of u and u' using av, i.e., $\kappa' =_{df} \mathrm{split}(\kappa, [u]_\kappa, av)$. Since $u \sim_\kappa u'$ but $u \not\sim_{\kappa'} u'$ (as $\kappa'(u)(av) \neq \kappa'(u')(av)$), we have $\mathcal{R}' \sqsubset \mathcal{R}$. ∎

In the above proof, an unclosed black-box abstraction $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ is refined by augmenting $\mathcal{U}$, and non-determinism is resolved by refining κ. The following corollary states that these are not only sufficient, but also necessary to eventually to re-establish the desired property (closedness or determinism).

Corollary 3.2

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction, and let $\mathcal{R}' = \langle \mathcal{U}', \kappa' \rangle$ be a (not necessarily strict) refinement of $\mathcal{R}$.

(i) If $\mathcal{R}$ is not closed and $\mathcal{U}' = \mathcal{U}$, then $\mathcal{R}'$ is not closed.

(ii) If $\mathcal{R}$ is not deterministic and $\kappa' = \kappa$, then $\mathcal{R}'$ is not deterministic.

Proof:

(i) Let $u \in \mathcal{U}, a \in \Sigma$ be such that $ua \not\sim_\kappa u'$ for all $u' \in \mathcal{U}$. Since $\kappa' \sqsubseteq \kappa$, this also implies that $ua \not\sim_{\kappa'} u'$ for all $u' \in \mathcal{U}$, hence $\mathcal{R}' = \langle \mathcal{U}, \kappa' \rangle$ is not closed.

(ii) Let $u, u' \in \mathcal{U}, a \in \Sigma$ be such that $u \sim_\kappa u'$, but $ua \not\sim_\kappa u'a$. Since u, u' are also elements of $\mathcal{U}' \supseteq \mathcal{U}$, and $\kappa = \kappa'$, the above assumption remains unaffected when considering $\sim_{\kappa'}$, hence $\mathcal{R}' = \langle \mathcal{U}', \kappa \rangle$ is not deterministic. ∎

It should be noted, however, that the refined black-box abstraction $\mathcal{R}'$ from the proof of Lemma 3.5 is not necessarily closed or deterministic. This then gives rise to yet another strict refinement $\mathcal{R}'' \sqsubset \mathcal{R}'$, and so on. As the number of classes in each black-box abstraction strictly increases, but cannot grow beyond $|\Sigma^*/\cong_\lambda|$ due to the validity of κ, Theorem 3.2 guarantees that, for a regular output function λ, the process eventually stabilizes with a closed and deterministic black-box abstraction.

As we have noted above, reachability and output inconsistencies also guarantee the existence of a strict refinement. In these cases, they also induce *counterexamples*, i.e., they can be analyzed to obtain a word $w \in \Sigma^*$ such that $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$.[6] In Section 3.3, we will see however that the more appropriate perspective is to view counterexamples as special cases of reachability or output inconsistencies, and how these phenomena can be analyzed in order to derive refined black-box abstractions (via the detour of introducing an unclosedness or non-determinism). Before looking into this, we will however first state how the "special" case of actual counterexamples may be exploited to refine a black-box abstraction.

Theorem 3.3

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a suffix-based black-box abstraction of an output function λ, and let $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$. Furthermore, let $w \in \Sigma^*$ be a counterexample, i.e., $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$. Then, the following two statements are true:

(i) w contains a prefix $\widehat{u}\widehat{a} \sqsubseteq_{pref} w$, $\widehat{u} \in \Sigma^*, \widehat{a} \in \Sigma$, such that $\mathcal{H}[\widehat{u}] = [\widehat{u}]_\kappa$, but $\mathcal{H}[\widehat{u}\widehat{a}] \neq [\widehat{u}\widehat{a}]_\kappa$, thus $\mathcal{R}' = \langle \mathcal{U} \cup \{\widehat{u}\}, \kappa \rangle$ is non-deterministic.

(ii) w can be decomposed into $w = \widehat{u}\widehat{a}\widehat{v}$, $\widehat{u}, \widehat{v} \in \Sigma^*, \widehat{a} \in \Sigma$ such that $\exists u \in \rho_{\mathcal{R}}(\mathcal{H}[\widehat{u}]) : \forall u' \in \rho_{\mathcal{R}}(\mathcal{H}[\widehat{u}\widehat{a}]) : \lambda(u\widehat{a}, \widehat{v}) \neq \lambda(u', \widehat{v})$, thus $\mathcal{R}' = \langle \mathcal{U}, \mathrm{split}(\kappa, \mathcal{H}[\widehat{u}\widehat{a}], \widehat{v}) \rangle$ is not closed.

At this point, we defer the proof to the next section. In particular, we will present two lemmas—Lemma 3.6 in Section 3.3.3 and Lemma 3.8 in Section 3.3.4—from which the proof for (i) and (ii) in the above theorem follows directly.

3.3. An Abstract Framework for Counterexample Analysis

In this section, we will prove the existence of the prefix and the decomposition of a counterexample with the properties stated in Theorem 3.3. Apart from proving their mere existence, we will also describe how they can be determined algorithmically. Notably, both cases can be reduced to instances of a more abstract problem, and solved with the very same approach.

We start by introducing our abstract framework for counterexample analysis, which is an extended and more flexible version of the one presented by Isberner and Steffen [108], and then describe how the problem of finding a prefix as mentioned in Theorem 3.3 (i) (*prefix-based* counterexample analysis, Section 3.3.3), and the problem of finding a decomposition according to Theorem 3.3 (ii) (*suffix-based* counterexample analysis, Section 3.3.4) can be formulated in this framework.

The general idea is shown in Figure 3.1: Theorem 3.3 states that a counterexample can be analyzed to either find a prefix or a decomposition, each satisfying certain properties. In both cases, the concrete counterexample is transformed to a common mathematical structure, called *abstract counterexample*, on which search algorithms can be applied to find a *breakpoint*. This breakpoint can then be used to derive either a prefix according to Theorem 3.3 (i), or a decomposition according to Theorem 3.3 (ii), depending on the source of the abstract counterexample.

[6] This counterexample is usually of the form $u \cdot v$, where $u \in \mathcal{U}\Sigma$ and $v \in Ch_\kappa(u)$.

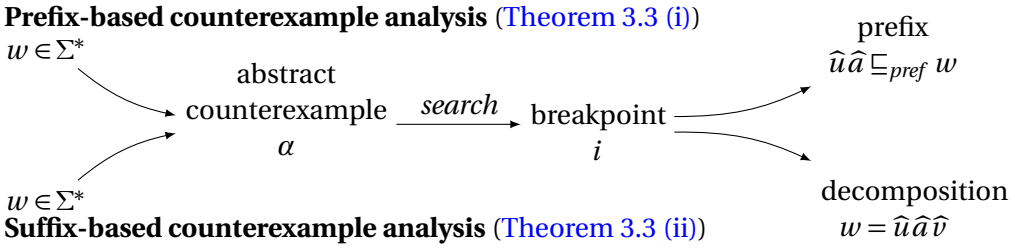


Figure 3.1.: Conceptual approach of abstract counterexample analysis applied to a concrete counterexample $w \in \Sigma^*$

3.3.1. Formal Definitions

We start by formally introducing the concept of an *abstract counterexample*, which for now we will treat as a merely syntactical entity, and leave it to the above-referenced subsections to establish a connection between a concrete counterexample $w \in \Sigma^*$ and its abstracted version. The intuition that indices in an abstract counterexample correlate in a certain way to positions in the corresponding concrete counterexample shall suffice at this point.

Definition 3.17 (*Abstract counterexample*)

An *abstract counterexample* is a quadruple $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$, where

- $\mathcal{E}$ is an arbitrary set (the *effect domain*),
- $\triangleright \subseteq \mathcal{E} \times \mathcal{E}$ is a *transitive* binary relation on $\mathcal{E}$ (the *effect relation*),
- $l \in \mathbb{N}^+$ is a positive integer, denoting the *length* of the abstract counterexample, and
- $\eta : \{0, \ldots, l\} \to \mathcal{E}$ is the *effect mapping*.

An abstract counterexample is called *valid* if and only if $\eta(0) \not\triangleright \eta(l)$.

The validity requirement is essential for guaranteeing the existence of a *breakpoint*, i.e., an index i such that $\eta(i)$ is not related (wrt. $\triangleright$) to its immediate successor $\eta(i+1)$. Again, we treat breakpoints as a purely syntactical concepts, with the intuition that breakpoints in the abstract counterexample allow to determine the prefix and the decomposition of the concrete counterexample, respectively.

Definition 3.18 (*Breakpoint*)

Let $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$ be an abstract counterexample. A *breakpoint* in α is an index i, $0 \leq i < l$, satisfying

$$\eta(i) \not\triangleright \eta(i+1).$$

Corollary 3.3

Let α be an abstract counterexample. If α is valid, then it contains a breakpoint.

Proof: Assume that $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$ is a valid abstract counterexample not containing a break-

Algorithm 3.2 Abstract counterexample analysis using binary search

Require: Valid abstract counterexample $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$
Ensure: Breakpoint $i, 0 \leq i < l$, satisfying $\eta(i) \not\triangleright \eta(i+1)$

```
1:  function BINARY-SEARCH_left(α)
2:      low ← 0, high ← l
3:      while (high − low) > 1 do                    ▷ Invariant: η(low) ⋫ η(high)
4:          mid ← ⌊(low + high)/2⌋
5:          if η(low) ⋫ η(mid) then
6:              high ← mid
7:          else                                     ▷ η(mid) ⋫ η(high) by transitivity
8:              low ← mid
9:          end if
10:     end while                                    ▷ Postcondition: η(low) ⋫ η(high) ∧ high = low + 1
11:     return low
12: end function
```

point, i.e.,

$$\forall 0 \leq i < l : \eta(i) \triangleright \eta(i+1).$$

By transitivity of $\triangleright$, we can conclude that then also $\eta(0) \triangleright \eta(l)$, contradicting the assumption that α was valid. ■

3.3.2. Finding Breakpoints

It is obvious that a breakpoint can be found using linear search, by scanning the indices of an abstract counterexample $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$ in ascending (descending) order and comparing each value of η to its immediate successor (predecessor). In fact, this even allows us to find the leftmost (rightmost) breakpoint; however, in the worst-case, η has to be evaluated at every single index.

A much better solution exists. Exploiting the transitivity of $\triangleright$ (which guarantees the existence of a breakpoint in an abstract counterexample in the first place), a *binary search* strategy can be employed: for indices $low, high$ satisfying $high - low > 1$ and $\eta(low) \not\triangleright \eta(high)$, any index $i, low < i < high$ will satisfy at least one of $\eta(low) \not\triangleright \eta(i)$ and $\eta(i) \not\triangleright \eta(high)$. The breakpoint search algorithm using binary search is given as Algorithm 3.2. Note that unlike in the case of a totally ordered search domain, there may be some degree of freedom regarding where to continue the search, as it is possible that both $\eta(low) \not\triangleright \eta(mid)$ as well as $\eta(mid) \not\triangleright \eta(high)$ hold. In this case, the search can be continued in either half, depending on whether breakpoints are *preferred* to be located near the left or the right.[7] Algorithm 3.2 prefers breakpoints towards the left end (hence the name BINARY-SEARCH$_{left}$). To obtain a version preferring breakpoints towards the right end, it is sufficient to replace the **if** condition in line 5 with $\eta(mid) \not\triangleright \eta(high)$, and swap the bodies of the **if** and the **else** blocks (lines 6 and 8, respectively).

We conclude the description of the abstract framework with the following proposition stating the complexity, and continue with describing instantiations of the framework.

[7] Note that binary search however cannot guarantee to find the left*most* or right*most* breakpoint.

Proposition 3.1

A breakpoint in an abstract counterexample $\alpha = \langle \mathcal{E}, \triangleright, l, \eta \rangle$ *can be found by evaluating* η *at no more than* $2 + \lceil \log l \rceil = \mathcal{O}(\log l)$ *different indices.*[8]

3.3.3. Prefix-based Counterexample Analysis

We will now describe how the problem of finding a prefix $\widehat{u}\widehat{a}$ of a counterexample Σ^* satisfying the conditions of Theorem 3.3 (i) can be reduced to finding a breakpoint in an abstract counterexample, i.e., the realization of the upper half of Figure 3.1. This comprises the *derivation* of an abstract counterexample from a concrete one, and the *translation* of a breakpoint in the abstract counterexample into the desired prefix.

First, however, we will elaborate on the mentioned aspect that counterexamples pose a special case of reachability inconsistencies.

Lemma 3.6

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ *be a closed and deterministic black-box abstraction of an output function* λ*, and let* $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$ *be the corresponding* DFA*.*

(i) If $w \in \Sigma^*$ *is a counterexample (i.e.,* $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$*), then* w *also constitutes a reachability inconsistency, i.e.,* $\mathcal{H}[w] \neq [w]_\kappa$*.*

(ii) If $w \in \Sigma^*$ *constitutes a reachability inconsistency, it contains a prefix* $\widehat{u}\widehat{a} \sqsubseteq_{pref} w$*,* $\widehat{u} \in \Sigma^*, \widehat{a} \in \Sigma$*, such that* $\mathcal{H}[\widehat{u}] = [\widehat{u}]_\kappa$*, but* $\mathcal{H}[\widehat{u}\widehat{a}] \neq \mathcal{H}[\widehat{u}\widehat{a}]$*.*

(iii) If $\widehat{u} \in \Sigma^*, \widehat{a} \in \Sigma$ *satisfy the conditions of (ii),* $\mathcal{R}' = \langle \mathcal{U} \cup \{\widehat{u}\}, \kappa \rangle$ *is not deterministic.*

Proof: At this point, we only prove (i) and (iii), and give a constructive proof for (ii) in the remainder of this section.

(i) Let $w \in \Sigma^*$ be a counterexample, i.e., $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$. As $\lambda_{\mathcal{H}}(w) = \lambda_{\mathcal{H}}^{\mathcal{H}[w]}(\varepsilon) \neq \lambda(w) = \lambda(w, \varepsilon) = \kappa(w, \varepsilon)$, we can conclude that $\mathcal{H}[w]$ and $[w]_\kappa$ must be distinct, as they are separated by ε.

(iii) Let $\widehat{u} \in \Sigma^*, \widehat{a} \in \Sigma$ be such that $\mathcal{H}[\widehat{u}] = [\widehat{u}]_\kappa$, but $\mathcal{H}[\widehat{u}\widehat{a}] \neq [\widehat{u}\widehat{a}]_\kappa$. Apparently, $\widehat{u} \notin \mathcal{U}$, as otherwise $\mathcal{H}[\widehat{u}\widehat{a}] = \delta_{\mathcal{H}}(\mathcal{H}[\widehat{u}], \widehat{a}) = [\widehat{u}\widehat{a}]_\kappa$. Thus, there exists $u \in \mathcal{U}$ such that $u \sim_\kappa \widehat{u}$ ($u \in \rho_{\mathcal{R}}(\mathcal{H}[\widehat{u}])$), but $u\widehat{a} \nsim_\kappa \widehat{u}\widehat{a}$ (as $[u\widehat{a}]_\kappa = \mathcal{H}[\widehat{u}\widehat{a}] \neq [\widehat{u}\widehat{a}]_\kappa$). As a result, $\mathcal{R}'$ will be non-deterministic. ∎

Let us now look at how finding a prefix according to Lemma 3.6 (ii) can be reduced to finding a breakpoint in an abstract counterexample.

[8] Unless otherwise noted, log denotes the binary logarithm ($\log_2$).

Definition 3.19

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction, and let $\mathcal{H} = \text{DFA}(\mathcal{R})$ be the corresponding DFA. The *derived abstract counterexample* of a word $w \in \Sigma^*$ is the abstract counterexample $\alpha = \langle \{0,1\}, \geq, |w|, \eta \rangle$, where

$$\eta \colon \{0,\ldots,|w|\} \to \{0,1\}, \;\; \eta(i) =_{df} \begin{cases} 0 \text{ if } \mathcal{H}[w_{1..i}] = [w_{1..i}]_\kappa \\ 1 \text{ otherwise} \end{cases} \quad \forall 0 \leq i \leq |w|.$$

Lemma 3.7

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction, let $\mathcal{H} = \text{DFA}(\mathcal{R})$ be its associated DFA*, and let $w \in \Sigma^*$ be a word constituting a reachability inconsistency, i.e., $\mathcal{H}[w] \neq [w]_\kappa$. Then, the derived abstract counterexample α for w as defined in Definition 3.19 is valid, and if i is a breakpoint in α, $\widehat{u} = w_{1..i}, \widehat{a} = w_{i+1}$ satisfy the conditions of Lemma 3.6 (ii).*

Proof: We first show that α is valid. Since $\mathcal{H}[\varepsilon] = [\varepsilon]_\kappa$ by definition, we have $\eta(0) = 0$. Furthermore, since w constitutes a reachability inconsistency, we have $\eta(|w|) = 1$. Since $0 \not\geq 1$, we can conclude that α is valid.

Let us now assume that $i, 0 \leq i < |w|$, is a breakpoint in α, i.e., $\eta(i) = 0$ and $\eta(i+1) = 1$. Let $\widehat{u} =_{df} w_{1..i}$ and $\widehat{a} =_{df} w_{i+1}$ (thus $\widehat{u}\widehat{a} = w_{1..i+1}$). Applying the definition of η, the breakpoint condition translates to $\mathcal{H}[\widehat{u}] = [\widehat{u}]_\kappa$ and $\mathcal{H}[\widehat{u}\widehat{a}] \neq [\widehat{u}\widehat{a}]_\kappa$, which directly correspond to the conditions stated in Lemma 3.6 (ii). ■

Remark 3.4

Lemma 3.6 (iii) provides "instructions" on how a black-box abstraction can be refined using the information from a reachability inconsistency, i.e., by adding the prefix $\widehat{u}$ to $\mathcal{U}$. The resulting short prefix set $\mathcal{U}'$ usually is not prefix-closed, which may introduce further reachability inconsistencies. However, these can be analyzed and exploited for refinement with the same analysis technique.

We conclude the description of prefix-based counterexample analysis[9] with an analysis of its complexity. We have already stated in Proposition 3.1 that, for an abstract counterexample of length m, η needs to be evaluated at $\mathcal{O}(\log m)$ different indices in order to find a breakpoint. Evaluating η as defined in Definition 3.19 requires evaluating κ on an arbitrary prefix of the counterexample, which in turn requires a number of membership queries equal to the size of the respective characterizing set. Let χ be the size of the largest characterizing set, then $\mathcal{O}(\chi \log m)$ queries are sufficient. We will see in the next chapter that, for a target DFA of size n, $\chi = \mathcal{O}(n)$ can be guaranteed, but also $\chi = \Theta(n)$ might be necessary. Thus, if the sizes of the characterizing sets are bounded by n, $\mathcal{O}(n \log m)$ membership queries are required.

How long are these membership queries? If the breakpoint is at position $m-1$ and binary search is used, all of the prefixes of w for which κ is evaluated have a length greater or equal to $m/2$. The length of the overall query depends on the length of the suffixes in the respective characterizing sets. Thus, if σ is the length of the longest suffix in any characterizing set, the combined number of symbols in all membership queries made during prefix-based counterexample analysis is in $\mathcal{O}\big((\chi \log m)(m+\sigma)\big)$. If the length of all suffixes as well as the size of the

[9] In this context, we will use the word "counterexample" to refer to *any* kind of reachability inconsistency.

characterizing sets are bounded by n, this can be simplified to $\mathcal{O}\big((n\log m)(n+m)\big)$, and—under the assumption that $m=\Omega(n)$—even further to $\mathcal{O}(nm\log m)$.

Proposition 3.2

If $\mathcal{R}=\langle\mathcal{U},\kappa\rangle$ is a black-box abstraction of a regular output function λ (with $n=_{df}|\Sigma^*/\cong_\lambda|$) satisfying

$$\forall u\in\Sigma^*:(|Ch_\kappa(u)|=\mathcal{O}(n)\wedge\forall v\in Ch_\kappa(u):|v|=\mathcal{O}(n)),$$

prefix-based analysis of a counterexample of length $m=\Omega(n)$ requires $\mathcal{O}(n\log m)$ membership queries that altogether contain $\mathcal{O}(nm\log m)$ symbols.

3.3.4. Suffix-based Counterexample Analysis

We will now consider suffix-based counterexample analysis. Here, the problem is to find a decomposition $w=\widehat{u}\widehat{a}\widehat{v}$ of a counterexample w satisfying the conditions of Theorem 3.3 (ii). This comprises describing how an abstract counterexample can be derived from a concrete one, and how a breakpoint in the abstract counterexample corresponds to the desired decomposition, thus realizing the lower half of Figure 3.1. Again, we first start by relating counterexamples to the more general concept of *output inconsistencies*.

Lemma 3.8

Let $\mathcal{R}=\langle\mathcal{U},\kappa\rangle$ be a closed and deterministic black-box abstraction of an output function λ, and let $\mathcal{H}=\mathrm{DFA}(\mathcal{R})$ be the corresponding DFA.

(i) If $w\in\Sigma^*$ is a counterexample (i.e., $\lambda_\mathcal{H}(w)\neq\lambda(w)$), then (ε,w) also constitutes an output inconsistency, i.e., $\lambda_\mathcal{H}^{[\varepsilon]_\kappa}(w)\neq\lambda(\varepsilon,w)$.

(ii) If $(x,y)\in\mathcal{U}\times\Sigma^*$ constitutes an output inconsistency, y can be decomposed into $y=\widehat{u}\widehat{a}\widehat{v}$, $\widehat{u},\widehat{v}\in\Sigma^*,\widehat{a}\in\Sigma$, such that

$$\exists u\in\rho_\mathcal{R}(\delta_\mathcal{H}([x]_\kappa,\widehat{u})):\forall u'\in\rho_\mathcal{R}(\delta_\mathcal{H}([x]_\kappa,\widehat{u}\widehat{a})):\lambda(u\widehat{a},\widehat{v})\neq\lambda(u',\widehat{v}).$$

(iii) If $(x,y)\in\mathcal{U}\times\Sigma^*$ constitutes an output inconsistency and $y=\widehat{u}\widehat{a}\widehat{v}$ is a decomposition of y satisfying the conditions of (ii), $\mathcal{R}'=\langle\mathcal{U},\mathrm{split}(\kappa,\delta_\mathcal{H}([x]_\kappa,\widehat{u}\widehat{a}),\widehat{v})\rangle$ is not closed.

Proof: Again, we only prove (i) and (iii), and give a constructive proof for (ii) in the remainder of this section.

(i) Let $w\in\Sigma^*$ be a counterexample, i.e., $\lambda_\mathcal{H}(w)\neq\lambda(w)$. As $\lambda_\mathcal{H}(w)=\lambda_\mathcal{H}(\varepsilon,w)=\lambda_\mathcal{H}^{q_{0,\mathcal{H}}}(w)\neq\lambda(\varepsilon,w)=\lambda(w)$ and $q_{0,\mathcal{H}}=[\varepsilon]_\kappa$ by definition, (ε,w) constitutes an output inconsistency.

(iii) Let $(x,y)\in\mathcal{U}\times\Sigma^*$ constitute an output inconsistency, and let $y=\widehat{u}\widehat{a}\widehat{v}$ and $u\in\rho_\mathcal{R}(\delta_\mathcal{H}([x]_\kappa,\widehat{u}))$ be chosen such that we have

$$\forall u'\in\rho_\mathcal{R}(\delta_\mathcal{H}([x]_\kappa,\widehat{u}\widehat{a})):\lambda(u\widehat{a},\widehat{v})\neq\lambda(u',\widehat{v}). \tag{3.1}$$

Note that this implies $u\widehat{a}\notin\mathcal{U}$, as otherwise $u\widehat{a}\in\rho_\mathcal{R}(\delta_\mathcal{H}([x]_\kappa,\widehat{u}\widehat{a}))$, and the universal quantification would not be valid.

Let $\kappa' =_{df} \text{split}(\kappa, \delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a}), \widehat{v})$. For any $u' \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a}))$, we know that $Ch_{\kappa'}(u') = Ch_{\kappa'}(u\widehat{a}) = Ch_\kappa(u') \cup \{\widehat{v}\}$ (as $u', u\widehat{a} \in \delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a})$), and furthermore that $\kappa'(u')(\widehat{v}) \neq \kappa'(u\widehat{a})(\widehat{v})$ due to (3.1). Thus, $u\widehat{a} \not\sim_\kappa u'$ for any $u' \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a}))$. $u\widehat{a} \in \mathcal{U}\Sigma$ furthermore cannot be $\sim_{\kappa'}$-equivalent to any other short prefix, as the set $\mathcal{U}$ has not changed and it was not $\sim_\kappa$-equivalent to any short prefix not in $\rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a}))$ (and thus also not $\sim_{\kappa'}$-equivalent, as $\sim_{\kappa'}$ refines $\sim_\kappa$). Therefore, κ' is not closed. ■

We now describe how determining a decomposition according to Lemma 3.8 (ii) can be reduced to finding a breakpoint in a corresponding derived abstract counterexample.

Definition 3.20

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction, and let $\mathcal{H} = \text{DFA}(\mathcal{R})$ be the corresponding DFA. The *derived abstract counterexample* of a pair $(x, y) \in \mathcal{U} \times \Sigma^*$ is the abstract counterexample $\alpha = \left\langle 2^{\mathbb{B}} \setminus \{\emptyset\}, \subseteq, |y|, \eta \right\rangle$, where the effect mapping η is defined as follows:

$$\eta : \{0, \ldots, |y|\} \to 2^{\mathbb{B}} \setminus \{\emptyset\}, \quad \eta(i) =_{df} \left\{ \lambda(u, y_{i+1..|y|}) \mid u \in \rho_{\mathcal{U}}(\delta_{\mathcal{H}}([x]_\kappa, y_{1..i})) \right\}.$$

Lemma 3.9

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction of some output function $\lambda : \Sigma^* \to \mathbb{B}$, let $\mathcal{H} = \text{DFA}(\mathcal{R})$ be its associated DFA*, and let $(x, y) \in \mathcal{U} \times \Sigma^*$ constitute an output inconsistency, i.e., $\lambda_{\mathcal{H}}^{[x]_\kappa}(y) \neq \lambda(x, y)$. Then, the derived abstract counterexample α as defined in Definition 3.20 is valid, and if i is a breakpoint in α, $\widehat{u} = y_{1..i}, \widehat{a} = y_{i+1}, \widehat{v} = y_{i+2..|y|}$ satisfy the conditions of Lemma 3.8 (ii).*

Proof: Again, we start by first showing that α is valid. Since ε is always in the characterizing set of any prefix in $\mathcal{U}$, we know that $\forall u, u' \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, y)) : \lambda(u, \varepsilon) = \kappa(u)(\varepsilon) = \kappa(u')(\varepsilon) = \lambda(u, \varepsilon)$. Thus, $\eta(|y|)$ is the singleton $\left\{ \lambda_{\mathcal{H}}^{[x]_\kappa}(y) \right\}$. Since (x, y) constitutes an output inconsistency, we know that $\lambda_{\mathcal{H}}^{[x]_\kappa}(y) \neq \lambda(x, y)$. As $\eta(0)$ contains $\lambda(x, y)$ (note that $x \in \mathcal{U}$, and therefore $x \in \rho_{\mathcal{R}}([x]_\kappa)$), we can conclude that $\eta(0)$ contains an element not in $\eta(|y|) = \left\{ \lambda_{\mathcal{H}}^{[x]_\kappa}(y) \right\}$. We therefore have established that $\eta(|y|)$ is a singleton and distinct from $\eta(0)$, thus $\eta(0) \not\subseteq \eta(|y|)$.

Let now $i, 0 \leq i < |y|$, be a breakpoint in α, i.e., $\eta(i+1)$ is a singleton and $\eta(i) \neq \eta(i+1)$. Let $\widehat{u} =_{df} y_{1..i}$, $\widehat{a} =_{df} y_{i+1}$ (thus $\widehat{u}\widehat{a} = y_{1..i+1}$), and $\widehat{v} =_{df} y_{i+2..|y|}$. As $\eta(i) = \left\{ \lambda(u, \widehat{a}\widehat{v}) \mid u \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u})) \right\}$ contains an element not in $\eta(i+1)$, and since $\lambda(u, \widehat{a}\widehat{v}) = \lambda(u\widehat{a}, \widehat{v})$, there needs to exist a $u \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}))$ such that $\lambda(u\widehat{a}, \widehat{v})$ is distinct from all values in $\left\{ \lambda(u', \widehat{v}) \mid u' \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a})) \right\} = \eta(i+1)$. This satisfies the condition from Lemma 3.8 (ii). ■

Remark 3.5

The presentation is much more complicated than in the original version due to Rivest and Schapire [155], as we consider the general case where there might be several representative short prefixes for each state in $\mathcal{H}$. If we can assume that for all $q \in Q_{\mathcal{H}}$ we have $|\rho_{\mathcal{R}}(q)| = 1$, and denoting the unique element of the set $\rho_{\mathcal{R}}(q)$ by $\lfloor q \rfloor$, the presentation becomes much simpler (and more intuitive): we can simply choose $\mathbb{B}$ as our effect domain, the equality relation as

the effect relation, and define the effect mapping as

$$\eta(i) =_{df} \lambda(\lfloor \delta_{\mathcal{H}}([x]_\kappa, y_{1..i})\rfloor, y_{i+1..|y|}).$$

The breakpoint condition then translates to

$$\lambda(\lfloor \delta_{\mathcal{H}}([x]_\kappa, \widehat{u})\rfloor \widehat{a}, \widehat{v}) \neq \lambda(\lfloor \delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a})\rfloor, \widehat{v}).$$

If furthermore $[x]_\kappa = q_{0,\mathcal{H}}$ (as is the case for the output inconsistency directly derived from a counterexample, cf. Lemma 3.8 (i)), this can be simplified to

$$\lambda(\lfloor \widehat{u}\rfloor_{\mathcal{H}} \widehat{a}, \widehat{v}) \neq \lambda(\lfloor \widehat{u}\widehat{a}\rfloor_{\mathcal{H}}, \widehat{v}),$$

where $\lfloor u\rfloor_{\mathcal{H}}$ is shorthand for $\lfloor \mathcal{H}[u]\rfloor$. This highlights that transforming the target of the $\widehat{a}$-transition of the state $\mathcal{H}[\widehat{u}]$ to its representative prefix in $\mathcal{U}$ changes the future behavior wrt. $\widehat{v}$, thus justifying the introduction of a new state as the target of this transition.

Remark 3.6

Lemma 3.8 (iii) again provides instructions on how the information obtained from analyzing an output inconsistency can be used to trigger refinement, namely by splitting the class $\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a})$ using $\widehat{v}$ as discriminator. This usually violates semantic suffix-closedness, which may result in further output inconsistencies. In analogy to Remark 3.4, these can however be dealt with using exactly the technique we just described.

Again, a note on the query and symbol complexities is in order. Evaluating η as defined in Definition 3.20 requires one membership query per element in $\rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, y_{1..i}))$. Thus, if r is the maximum number of representatives per class in $\mathscr{C}(\mathcal{R})$, $\mathcal{O}(r \log m)$ membership queries are required, where $m = |y|$. It can easily be ensured that every class in $\mathscr{C}(\mathcal{R})$ has a unique representative in $\mathcal{U}$ (the algorithm by Rivest and Schapire [155] accomplishes this, for instance). In this case, the number of membership queries reduces to $\mathcal{O}(\log m)$.

Let us now consider the symbol complexity. The length of the suffix in the membership queries of the form $\lambda(u, y_{i+1..|y|})$ is only bounded by m. If ℓ is the maximum length of any prefix in $\mathcal{U}$, then $m+\ell$ is an upper bound for the length of each query. Thus, the total number of symbols in all queries during the suffix-based analysis of a counterexample is $\mathcal{O}\big((m+\ell) r \log m\big)$. It is furthermore easy to ensure that no prefix in $\mathcal{U}$ is longer than n, which—in conjunction with assuming that each class has a unique representative—allows us to simplify the symbol complexity to $\mathcal{O}\big((n+m)\log m\big)$, and further to $\mathcal{O}(m \log m)$ under the additional assumption that $m = \Omega(n)$.

Proposition 3.3

If $\mathcal{R} = \langle \mathcal{U}, \kappa\rangle$ is a black-box abstraction of a regular output function λ (with $n =_{df} |\Sigma^*/\cong_\lambda|$) satisfying

$$|\mathscr{C}(\mathcal{R})| = |\mathcal{U}| \wedge \forall u \in \Sigma^* : |u| = \mathcal{O}(n),$$

suffix-based analysis of a counterexample of length $m = \Omega(n)$ requires $\mathcal{O}(\log m)$ membership queries that altogether contain $\mathcal{O}(m \log m)$ symbols.

3.3.5. Improved Search Strategies

Isberner and Steffen [108] have observed that binary search, while guaranteeing a logarithmic worst-case complexity for finding breakpoints, suffers from the disadvantage that long counterexamples inevitably lead to long queries: for instance, assuming that $w \in \Sigma^*$ constitutes a reachability inconsistency, binary search (cf. Algorithm 3.2) will first evaluate the effect mapping η of the corresponding abstract counterexample at index $\lfloor m/2 \rfloor$, where $m = |w|$, corresponding to classifying $w_{1..\lfloor m/2 \rfloor}$. Thus, the length of the first query is at least $\lfloor m/2 \rfloor$, which might be a problem if m is excessively long.

When analyzing reachability inconsistencies, it is generally preferable to evaluate η at low indices, as this corresponds to shorter queries. To realize this while maintaining a logarithmic worst-case complexity, Isberner and Steffen [108] propose to use *exponential search* instead of binary search, i.e., evaluating $\eta(0)$, $\eta(2^0) = \eta(1)$, $\eta(2^1) = \eta(2)$, etc., until $\eta(2^i) \not\simeq \eta(2^{i+1})$, and then using binary search (preferring breakpoints to the left) to find a breakpoint between indices 2^i and 2^{i+1}. In the worst case, this requires roughly twice as many queries (i.e., evaluations of η) as binary search, but in practice the number of queries is often much lower, and these queries are furthermore shorter. The experimental evaluation [108] suggests that exponential search results in almost the shortest prefixes (second only to linear search, which guarantees finding shortest prefixes but requires the highest number of queries), while requiring the lowest number of both queries and symbols of all considered approaches.

The same considerations also apply to the analysis of output inconsistencies $(x, y) \in \mathcal{U} \times \Sigma^*$. There, however, queries are of the form $\lambda(u, y_{i+1..|y|})$, where $u \in \mathcal{U}$. This means that evaluating η at *higher* indices corresponds to shorter queries (at least shorter suffixes, but $u \in \mathcal{U}$ is generally assumed to be rather short, compared to y). Thus, the direction of the exponential search needs to be reversed (considering $\eta(l-2^0)$, $\eta(l-2^1)$ etc.), and binary search needs to be adapted to prefer breakpoints to the right (cf. Section 3.3.2).

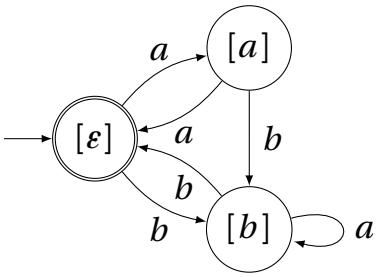
3.3.6. Comparison

In the previous sections, we have shown that both prefix- and suffix-based counterexample analysis (or, more generally, analysis of both reachability and output inconsistencies) can be reduced to a common problem: finding a breakpoint in an abstract counterexample. This breakpoint determines the prefix $\widehat{u}$ in the case of prefix-based analysis, and the decomposition $\widehat{u}\widehat{a}\widehat{v}$ in the case of suffix-based analysis. The prefix or decomposition can be used to obtain a refined black-box abstraction $\mathcal{R}'$, by violating determinism or closedness, respectively. Restoring these as described in the proof of Lemma 3.5 may violate reachability or output consistency, which can however be restored by repeated applications of the respective counterexample analysis method.

The symmetry breaks when the cost of the respective analysis is considered. If short prefixes in $\mathcal{U}$ are kept $\sim_\kappa$-inequivalent (which is rather easy to accomplish), finding a breakpoint for suffix-based analysis is possible using $\mathcal{O}(\log m)$ queries, where m is the length of the counterexample. On the other hand, there are regular output functions λ where characterizing sets of size $\Theta(n)$ are necessary ($n = |\Sigma^*/\cong_\lambda|$). Thus, prefix-based analysis in these cases requires $\mathcal{O}(n \log m)$ queries. Finding an intuitively accessible reason why suffix-based counterexample analysis is inherently less expensive than its prefix-based counterpart remains an open problem.

		$\mathcal{V}$: ε	a
$\mathcal{U}$	ε	1	0
	a	0	1
	b	0	0
	bb	1	0
$\mathcal{U}\Sigma\setminus\mathcal{U}$	aa	1	0
	ab	0	0
	ba	0	0
	bba	0	1
	bbb	0	0

(a) Sample observation table



(b) Automaton constructed from observation table on the right

Figure 3.2.: Example observation table and corresponding automaton during a run of L* [19]

3.4. Realizations

In this section, we will show how many existing learning algorithms can be viewed as instantiations of the described framework. Most learning algorithms vary in two aspects: the *data structures* they use for realizing a black-box abstraction and storing observations, and how they *handle counterexamples*. We will thus first introduce the two prevalent data structures, discuss how they realize black-box abstractions in the sense of this chapter, and use these to sort existing active automata learning algorithms into groups. Within these groups, we will then discuss how each algorithm handles counterexamples, and how this handling relates to the approaches presented in Section 3.3.

Note that the above only applies to learning algorithms that can in some way be regarded as *descendants* of L*. We will explicitly not cover learning algorithms that take an entirely different approach, i.e., that are not based on an over-approximation of the Nerode congruence (such as the CGE algorithm by Meinke [132]), or that do not aim at inferring canonical DFAs (such as NL* by Bollig *et al.* [36]).

3.4.1. Data Structures

Generally, there are two prevalent data structures used in an active automata learning context: *observation tables* and *discrimination trees*. These are typically used for storing information on how to realize the black-box classifier κ, as well as storing the short prefix set $\mathcal{U}$.

Observation tables. Perhaps the most famous data structure to be used in the context of active automata learning is the *observation table*. Originally introduced by Gold [75], it forms the central data structure of the first efficient active automata learning algorithm L*, presented by Angluin [19]. Variants of the observation table are used in a number of active automata learning algorithms for other machine types as well.

An example observation table, derived from querying the DFA from Figure 2.2a, is shown in Figure 3.2a. Both rows and columns of the table are indexed with words from Σ^*. Furthermore, the table is split in two parts: in the upper part, rows are indexed with prefixes from

$\mathcal{U} = \{\varepsilon, a, b, bb\}$, corresponding to states in the constructed DFA (shown in Figure 3.2b). As constructing the DFA requires determining the equivalence class of ua for each $u \in \mathcal{U}, a \in \Sigma$, the rows in the lower part correspond to those prefixes in $\mathcal{U}\Sigma$ that are not present in the upper part.

The cell corresponding to a row labeled by u and a suffix labeled by v stores the observation $\lambda(u, v)$. Hence, an observation table realizes a *global* suffix-based classifier (cf. Definition 3.7), as each of the suffixes in the global suffix set $\mathcal{V}$, which is given by the column labels, applies to all prefixes that are used as row indices.

Two prefixes are determined to be equivalent if the contents of the corresponding rows are equal. Thus, as the row labeled by ab in the lower part of the table contains the same values as the row in the upper part labeled by b, $[b]$ is chosen as the b-successor of $[a]$ in Figure 3.2b. For the same reason, there is no separate state for $[bb]$, as it is determined to be equivalent to ε. Finally, $[\varepsilon]$ is accepting in the corresponding DFA since the value in the column labeled by ε is 1.

To realize *splitting* of classes, a column with the new suffix is added to the table, and the newly introduced cells are filled using membership queries. Augmenting $\mathcal{U}$ is achieved by adding new rows to the upper part of the table (and thus also to the lower part). In the frequently occurring case that the new row label is in $\mathcal{U}\Sigma \setminus \mathcal{U}$ (e.g., for fixing an unclosedness as described in the proof of Lemma 3.5), this process is better described as *moving* a row from the lower to the upper part of the table (and adding new rows in the lower part).

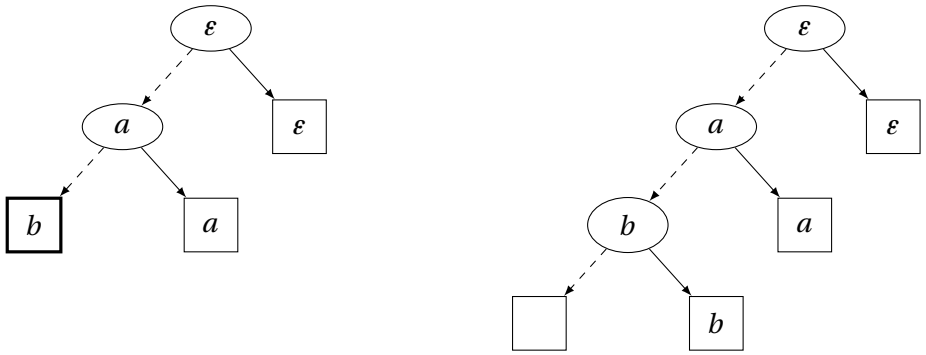
Discrimination trees. The observation table data structure is intuitive and easy to visualize, but contains some inherent redundancy (apart from the superfluous short prefix bb in the above example): usually not all suffixes in $\mathcal{V}$ are necessary to distinguish the equivalence classes. For example, the suffix ε alone is sufficient to distinguish $[\varepsilon]$ from the other two classes.

Kearns and Vazirani [115] proposed to realize the classification using a decision tree, which we will refer to as *discrimination tree*. An example for such a discrimination tree is shown in Figure 3.3a, for the same target DFA as the above observation table.[10] The process of classifying a prefix $u \in \Sigma^*$ using a discrimination tree can informally be described as follows: starting at the root node, whenever the current node is an inner node (elliptical shape) labeled with a *discriminator* $v \in \Sigma^*$, $\lambda(u, v)$ is evaluated. We then proceed to the 0-child (dashed arrow) or to the 1-child (solid arrow), depending on the observed outcome. This process is repeated until we finally reach a leaf (rectangular shape), which is labeled with the representative short prefix(es) $u' \in \mathcal{U}$, determining the class $[u']$ of u. The whole process of moving from the root to a leaf in this fashion is referred to as *sifting* u into the tree.

The set of short prefixes is given by the set of all leaf labels, i.e., $\mathcal{U} = \{\varepsilon, a, b\}$ in Figure 3.3a. The characterizing set for a prefix $u \in \Sigma^*$ is exactly the set of discriminators encountered at inner nodes on the path from the root to the tree, i.e., $Ch(\varepsilon) = \{\varepsilon\}$, and $Ch(a) = Ch(b) = \{\varepsilon, a\}$. Another useful property of a discrimination tree is that for two inequivalent prefixes $u, u' \in \Sigma^*$, there is guaranteed to be a single *separator*, which is the label of the *lowest common ancestor* of the corresponding leaves reached when sifting u and u', respectively (this will be explained in more detail in Section 4.1.2).

It should be noted that a discrimination tree does not store information about the classes of the successors of states identified by short prefixes in $\mathcal{U}$—in contrast to an observation table, where the lower part exists for precisely this reason. Thus, whenever a hypothesis $\mathcal{H}$ is con-

[10] We have omitted the superfluous short prefix bb that was part of the observation table from Figure 3.2a. While it would be possible to permit leaves with multiple short prefix labels, this is typically avoided in existing algorithms.



(a) Discrimination tree resulting in the DFA from Figure 3.2b

(b) Discrimination tree after splitting leaf labeled with b

Figure 3.3.: Example discrimination trees

structed, every word $ua \in \mathcal{U}\Sigma$ needs to be sifted into the tree (which requires a lot of additional membership queries), or this information has to be stored in a separate data structure. We will give a detailed description on how this can be accomplished efficiently in Section 4.2.2.

Splitting classes in a discrimination tree is usually accomplished by *splitting leaves*, i.e., replacing a leaf with an inner node with two children (leaves). Figure 3.3b shows the result of splitting the leaf labeled with b (and drawn with a thick border) in Figure 3.3a using b as discriminator.[11] This results in a leaf with no label, which calls for augmenting $\mathcal{U}$. This is in turn accomplished by attaching a label to the unlabeled leaf.

A notable aspect about splitting leaves in a discrimination tree is that it results in the *least refined* classifier that accommodates the new suffix for the class to be split: if κ is the classifier associated with the discrimination tree from Figure 3.3a, then the associated classifier of Figure 3.3b is $\kappa' = \text{split}(\kappa, [b]_\kappa, b)$ (cf. Definition 3.16), which in turn is the least refined classifier that both refines κ and satisfies $b \in Ch_{\kappa'}(b)$. This distinguishes discrimination trees from observation tables, where splitting classes is realized by adding suffixes to $\mathcal{V}$, resulting in an even more refined classifier (cf. Remark 3.3).

Other approaches. Most algorithms that infer canonical DFAs by over-approximating the Nerode congruence choose to store their observations and classification results in one of the above two data structures. An exception is the DHC algorithm by Merten *et al.* [138, 140]: it realizes a black-box abstraction merely in terms of a global black-box classifier κ that is determined by the global suffix set $\mathcal{V}$. The set $\mathcal{U}$ is computed dynamically as a minimal prefix-closed set containing ε, resulting in a closed (in the sense of Definition 3.9) black-box abstraction. The computation is typically performed in a breadth-first fashion, as sketched in Algorithm 3.3. An undesirable side-effect of this approach is that the set $\mathcal{U}$ does not necessarily grow monotonically. Hence, subsequent black-box abstractions might not be refinements of each other in the sense of Definition 3.15.

[11] The fact that the discriminator is the same as the leaf label is pure coincidence.

Algorithm 3.3 Dynamic computation of $\mathcal{U}$ (given κ) in a breadth-first fashion

Require: Black-box classifier $\kappa \in K_\lambda$
Ensure: Prefix-closed set $\mathcal{U}$ such that $\langle \mathcal{U}, \kappa \rangle$ is closed and deterministic

```
1:  U ← {ε}
2:  Q ← init_queue(ε)                          ▷ initialize new queue containing ε
3:  while Q ≠ ∅ do
4:      u ← poll(Q)                            ▷ retrieve and remove first element in Q
5:      for a ∈ Σ do
6:          if ∄u' ∈ U : ua ~_κ u' then        ▷ found closedness violation
7:              U ← U ∪ {ua}
8:              add(Q, ua)                     ▷ enqueue ua
9:          end if
10:     end for
11: end while
```

3.4.2. Handling Counterexamples

The choice of data structures is probably the most distinguishing characteristic between active learning algorithms, whereas the approach to handling counterexamples is much more subtle (in fact, some algorithms even treat counterexample handling as a "plug-in"). In the following, we thus list existing counterexample handling strategies separately for each data structure.

Observation table-based algorithms. These form the majorities of active automata learning algorithms that have been described. The comparably large number of different strategies is due to heuristic approaches that maintain *both* prefix-closedness of $\mathcal{U}$ and suffix-closedness of $\mathcal{V}$, while attempting to keep the overhead low.

Classical L* [19] When presented with a counterexample $w \in \Sigma^*$, all elements of $\mathrm{Pref}(w)$ are added to $\mathcal{U}$ (i.e., to the upper part of the table). This includes the prefix $\widehat{u} \sqsubset_{pref} w$ according to Theorem 3.3 (i), and thus—in conjunction with Corollary 3.2 (ii), which is applicable since $\mathcal{V}$ remains unchanged—causes non-determinism.

L^*_{col} ("Maler/Pnueli") [127] Even though Maler and Pnueli [127] presented an algorithm for inferring a subclass of Büchi automata, their strategy of adding all suffixes of a counterexample $w \in \Sigma^*$ to $\mathcal{V}$ has been adapted to the DFA case and is often referred to as L^*_{col}. One of these suffixes is $\widehat{v}$ satisfying the conditions of Theorem 3.3 (ii). Adding it to $\mathcal{V}$ causes all classes, including $\mathcal{H}[\widehat{u}\widehat{a}]$, to be split, which in conjunction with Corollary 3.2 (i) results in an unclosedness.

Shahbaz's algorithm [161] A counterexample w is decomposed into $w = u \cdot v$ such that $u \in \mathcal{U}\Sigma$ and u is of maximal length (over all possible decompositions satisfying this constraint). This is justified by the fact that the algorithm maintains $\mathcal{U}$ as a prefix-closed set, and the decomposition according to Theorem 3.3 (ii) necessarily implies $\widehat{u}\widehat{a} \notin \mathcal{U}$ (as otherwise $\rho_{\mathcal{R}}(\mathcal{H}[\widehat{u}\widehat{a}]) \cap \rho_{\mathcal{R}}(\mathcal{H}[\widehat{u}]) \cdot \{\widehat{a}\} \neq \emptyset$). Thus, under these circumstances, the suffix $\widehat{v}$ must be a suffix of v. Adding all elements in $\mathrm{Suff}(v)$ to $\mathcal{V}$ causes an unclosedness for the same reasons as the above strategy.

Suffix1by1 [106] Presented with a counterexample $w \in \Sigma^*$, elements of $\mathrm{Suff}(w)$ are added to $\mathcal{V}$ one by one, in ascending order of their lengths. This process is repeated until the table is no longer closed. Theorem 3.3 (ii) guarantees that a suffix causing an unclosedness is eventually encountered, however, Suffix1by1 does not guarantee to add a suffix satisfying the conditions of Theorem 3.3 (ii), as an unclosedness may be caused merely by coincidence. Still, it is ensured that an unclosedness with subsequent refinement occurs.

Rivest and Schapire [155] Using binary search, a *single* suffix satisfying the conditions of Theorem 3.3 (ii) is determined and added to $\mathcal{V}$ (which thus is *not* maintained as suffix-closed), resulting in an unclosedness and subsequent refinement.

Discrimination-tree based algorithms. For this class of algorithms, there are basically only two documented approaches that can be found in the literature. This is probably due to the fact that maintaining suffix-closedness, which is the main point of many of the heuristics for observation table-based algorithms, is not trivially possible in a discrimination tree without violating the property of each inner node having at least two children.

Kearns and Vazirani [115] Given a counterexample $w \in \Sigma^*$, a prefix $\widehat{u}\widehat{a} \sqsubseteq_{pref} w$ satisfying the conditions of Theorem 3.3 (i) is determined using linear search (a binary or exponential search strategy was proposed by Isberner and Steffen [108]), and $\widehat{u}$ is added to $\mathcal{U}$. This causes non-determinism, which is immediately resolved by splitting the leaf corresponding to $\widehat{u}$, using $\widehat{a} \cdot v$ as discriminator. Here, $v \in Seps_\kappa(\mathcal{H}[\widehat{u}\widehat{a}], [\widehat{u}\widehat{a}]_\kappa)$ is a suffix separating $[\widehat{u}\widehat{a}]_\kappa$ and $\mathcal{H}[\widehat{u}\widehat{a}]$.

Observation Pack [93] Given a counterexample $w \in \Sigma^*$, a decomposition $\widehat{u}\widehat{a}\widehat{v} = w$ satisfying the conditions of Theorem 3.3 (ii) is determined using binary search, and $\widehat{v}$ is used to split the class $\mathcal{H}[\widehat{u}\widehat{a}]$. This results in an unclosedness, which is immediately resolved by adding $\lfloor \widehat{u} \rfloor_{\mathcal{H}} \cdot \widehat{a}$ to $\mathcal{U}$, where $\lfloor \widehat{u} \rfloor_{\mathcal{H}}$ denotes the unique representative of $\mathcal{H}[\widehat{u}]$ (cf. Remark 3.5).

3.4.3. Complexity Considerations

Even though the asymptotic complexities (according to the measures described in Section 3.2.1) are specific to each algorithm, some lower bounds can be established for those algorithms that can be regarded as instances of the described framework. Table A.1 in Appendix A provides an overview.

It is obvious that at most $n-1$ equivalence queries are required, as each counterexample wrt. a black-box abstraction $\mathcal{R}$ can be exploited for refinement (cf. Theorem 3.3), resulting in a strict increase of $|\mathscr{C}(\mathcal{R})|$. However, Theorem 3.2 guarantees that once $|\mathscr{C}(\mathcal{R})| = n$, the hypothesis is necessarily correct.

Query complexity. Let us now consider the membership queries that are necessary to construct the final hypothesis (i.e., assuming that $\mathcal{U}$ and κ are given such that $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ satisfies $|\mathscr{C}(\mathcal{R})| = n$). Of course, queries need to be asked for constructing the intermediate hypotheses as well, but due to the usually monotonic growth of $\mathcal{U}$ and the monotonic refinement of κ, this information can be reused and does not need to be considered separately. Constructing the final hypothesis requires determining the classes wrt. $\sim_\kappa$ of all elements in $\mathcal{U} \cup \mathcal{U}\Sigma$. If χ is the maximum size of any characterizing set (i.e., $\chi = \max_{u \in \mathcal{U}} |Ch_\kappa(u)|$), this requires $\mathcal{O}(|\mathcal{U}| \cdot k \cdot \chi)$ membership queries.

It is easy to ensure $|\mathcal{U}| = n$ (i.e., by maintaining $\mathcal{U}$ as a set of pairwisely inequivalent prefixes). If we could ensure $\chi = \mathcal{O}(n)$, this would in combination result in an overall membership query complexity of $\mathcal{O}(kn^2)$, which coincides with the known lower bound for the problem, proven by Balcázar *et al.* [25]. However, to achieve $\chi = \mathcal{O}(n)$, the characterizing sets may be augmented by no more than a constant number of suffixes with each counterexample. This cannot be ensured by algorithms that add all suffixes of (a suffix of) a counterexample to the characterizing sets, such as the one by Maler and Pnueli [127], Shahbaz's algorithm [161], or Suffix1by1 [105, 106]: these can only guarantee $\chi = \mathcal{O}(nm)$, resulting in an overall query complexity of $\mathcal{O}(kn^2m)$, which intersects $\Omega(kn^3)$ under the assumption $m = \Omega(n)$.[12] The original L* algorithm [19] faces a similar problem: adding all prefixes of a counterexample to $\mathcal{U}$ results in $|\mathcal{U}| = \mathcal{O}(nm)$ while ensuring $\chi \leq n$, resulting in the same worst-case complexities.

The discrimination tree-based algorithms [93, 115], as well as the one by Rivest and Schapire [155], maintain unique representatives in $\mathcal{U}$, and only augment (some or all) characterizing sets by a single suffix per counterexample, thus ensuring $|\mathcal{U}| \leq n, \chi \leq n$ and resulting in a query complexity in $\mathcal{O}(kn^2)$ for hypothesis construction. However, *counterexample analysis*, i.e., determining either the prefix to add to $\mathcal{U}$ or the suffix to use for refinement, requires additional membership queries, resulting in query complexities in $\mathcal{O}(kn^2 + n\log m)$ (Rivest and Schapire, Observation Pack; cf. Proposition 3.3) or $\mathcal{O}(kn^2 + n^2\log m)$ (Kearns and Vazirani; cf. Proposition 3.2). Note that especially the former query complexity is almost optimal, and in fact coincides with the known lower bound if the length of counterexamples satisfies $m = 2^{\mathcal{O}(nk)}$. It remains an open problem to show that either $\Omega(kn^2 + n\log m)$ is the actual lower bound for the problem, or to give an algorithm that achieves a query complexity in $\mathcal{O}(kn^2)$ regardless of the length of counterexamples.[13]

Symbol complexity. Obtaining the corresponding symbol complexities is fairly easy: note that all of the mentioned algorithms either add *prefixes* of a counterexample to $\mathcal{U}$, or *suffixes* of a counterexample to the characterizing sets. This means that either the length of the prefixes is bounded by m and those of the suffixes by n, or vice versa. Since every query for hypothesis construction is composed of a prefix $u \in \mathcal{U} \cup \mathcal{U}\Sigma$ and a suffix $v \in Ch_\kappa(u)$, the length of each of this queries is in $\mathcal{O}(n+m)$, or $\mathcal{O}(m)$ under the assumption $m = \Omega(n)$. Thus, the symbol complexities for the algorithms can be obtained by multiplying the above asymptotic query complexities by m (or $n+m$).

3.5. Adaptation for Mealy Machines

The presented framework is already very general, in the sense that it does not rely on specifics of the DFA learning scenario, such as, e.g., that the output domain $\mathcal{D} = \mathbb{B}$ contains only two values.[14] Generalizing the concepts to variants of DFAs with larger (yet finite) output domains is straightforward. An example are *three-valued* DFA*s* (3DFAs), that augment the usual set of outputs—

[12] It should be noted that the estimate is very pessimistic (but nonetheless realizable), as adding a large number of suffixes to $\mathcal{V}$ table usually results in many additional states, thus reducing the overall number of required counterexamples.

[13] A possible approach would be to show that counterexamples of length $m = 2^{\omega(kn)}$ contain so much inherent redundancy that it is possible to reduce them to counterexamples of length $2^{\mathcal{O}(kn)}$ using at most $\mathcal{O}(kn)$ membership queries per counterexample. This would guarantee an asymptotic overall query complexity of $\mathcal{O}(kn^2)$. However, it remains unclear how this could be accomplished without the learner knowing the true value of n.

[14] An exception to this is, of course, the construction of the hypothesis DFA according to Definition 3.10.

"accept" and "reject"—with a third one, corresponding to "don't care". A learning algorithm for 3DFAs was presented by Chen *et al.* [47].

These generalizations are often used as a blueprint for developing automata learning algorithms for actively inferring finite-state transducers: it is often stated (e.g., by Hopcroft and Ullman [90] in the first edition of their classic book) that DFAs essentially are *Moore machines* [142] with a binary output alphabet. This perspective might be adequate in a "white-box" context, but not in the *black-box* context of active learning: if $\lambda : \Sigma^* \to \Omega^*$ is the output function of a Moore machine (or any other *letter-to-letter* transducer, cf. Remark 2.2), then, for words $w, w' \in \Sigma^*$ such that we have $w \sqsubset_{pref} w'$, evaluating $\lambda(w')$ yields strictly more information than evaluating $\lambda(w)$. This is unlike the case of (multi-valued) DFAs, where it is not possible to deduce the value of $\lambda(w)$ from knowing $\lambda(w')$, or vice versa.[15] Merely considering the last symbol of an output word, as was done in the first active automata learning algorithm for Mealy machines by Niese [129, 146], means discarding potentially valuable information, which is inexcusable for an algorithm that is meant to be efficient.

This calls for treating transducers as a separate class of target systems, rather than as a special case of multi-valued DFAs. Between the prevalent—and, to some extent,[16] equi-expressive—models of Mealy and Moore machines, the former are clearly the more desirable ones to consider, as they may be smaller than the latter by a factor of the size of the output alphabet. The first active automata learning algorithm for Mealy machines, L^*_M, is due to Niese [129, 146], and the description was later improved and formalized by Shahbaz and Groz [161]. In this section, we will sketch how the described framework can be adapted to cover the learning of Mealy machines. We will see that the consideration on a strictly formal level exposes considerably more differences than the comparison between L^* and L^*_M (which is a relatively straightforward adaption of the former) would suggest, which is due to the fact L^*_M is often used in conjunction with a number of heuristics that ensure a successful termination at the cost of additional membership queries.

3.5.1. Black-Box Abstractions for Mealy Machines

The concept of black-box abstractions (cf. Definition 3.8) remains mostly unchanged in the context of learning a Mealy machine model for an output function $\lambda : \Sigma^* \to \Omega^*$, where $\lambda = \lambda_{\mathcal{M}}$ is the output function of some canonical ("target") Mealy machine $\mathcal{M}$. This is mostly due to the fact that the relation $\cong_\lambda$ as defined in Definition 3.2 has the same characteristics as for DFAs: its index is finite if and only if there exists a Mealy machine computing λ, and the equivalence classes in $\Sigma^*/\cong_\lambda$ furthermore correspond to the states of the *canonical* Mealy machine for λ [167].

Thus, we use a finite set $\mathcal{U} \subset \Sigma^*$ of *short prefixes* to represent equivalence classes of a relation induced by a black-box classifier κ. Reflecting the fact that the output domain is $\mathcal{D} = \Omega^*$, the

[15] There may exist cases where precisely this is possible, depending on the observed values: if the language of the target DFA is known to be *prefix-closed*, $\lambda(w') = 1$ implies $\lambda(w) = 1$, and $\lambda(w) = 0$ implies $\lambda(w') = 0$. However, such *domain-specific knowledge* is usually better incorporated using optimizing *filters*, as described by, e.g., Margaria *et al.* [130].

[16] There exist two slightly different definitions of Moore machines: one where the *current* state determines the output (this were the semantics originally intended by Moore [142]), and one where the *successor* state determines the output. In the former case, the first output symbol is always fixed, thus there might exist Mealy machines that cannot be translated into a Moore machine of this kind (unless the first output symbol is discarded). The latter interpretation is truly equi-expressive with Mealy machines, but lacks a canonical form, as there might be several possible choices for the initial state due to its output not being observable.

black-box classifier κ is now defined as a function mapping *prefixes* $u \in \Sigma^*$ to partial functions (with finite domains) from Σ^* to Ω^*. Hence, formally we have

$$\kappa : \Sigma^* \to \left\{ f : \Sigma^* \rightharpoonup \Omega^* \mid |\operatorname{dom} f| < \infty \right\}.$$

The validity requirement remains unchanged as well: for $u \in \Sigma^*$ and $v \in Ch_\kappa(u)$, we demand that $\kappa(u)(v) = \lambda(u,v)$, exploiting the property of *suffix-observability* of Mealy machine output functions, i.e., $\lambda(u,v) = \lambda(u \cdot v)_{|u|+1..|u|+|v|}$. If the above validity requirement is satisfied, it is guaranteed that $\sim_\kappa$ is refined by the relation $\cong_\lambda$.

Remark 3.7 (*Terminology*)

Shu and Lee [162] introduced the term *output query* for the equivalent of a membership query in the setting of learning transducers, which was subsequently adapted by Shahbaz and Groz [161] in their description of L^*_M. However, in this thesis—and in accordance with the original L^*_M description given by Niese [129, 146]—we will use the term "membership query" to denote the evaluation of *any* (suffix-observable) output function $\lambda(\cdot,\cdot)$, since it refers to the same concept regardless of whether the target FSM is a DFA or a Mealy machine. It should be noted that this choice is made for historical reasons only, as "output query" is admittedly the more general term.

Transition Outputs

In a Mealy machine $\mathcal{M}$ (cf. Definition 2.7), the concept of accepting or rejecting states is replaced with *transition outputs*, i.e., a mapping $\gamma_\mathcal{M} : Q_\mathcal{M} \to \Omega$. We have remarked in Section 2.2.5 that these constitute the *local property* of states in a Mealy machine, that, e.g., has to be preserved by an isomorphism. In the case of DFA learning, we have ensured the correctness of the local property "acceptance" by enforcing that ε is always a member of the characterizing sets.

A comparable approach in the context of learning Mealy machines would be to require $\Sigma \subseteq Ch_\kappa(u)$ for all $u \in \Sigma^*$.[17] In fact, L^*_M does just that by initializing the global suffix set as $\mathcal{V} = \Sigma$. Irfan *et al.* [107] have observed that this scales poorly for large alphabet sizes. Furthermore, this cannot be realized when using a discrimination tree data structure without relaxing the requirement that each inner node must have at least two children. However, as ε has no discriminatory power in the context of Mealy machines, there is no "natural" choice for a minimum subset of all characterizing sets other than $\emptyset$.

Irfan *et al.* [107] proposed the algorithm L_1, which is a modification of L^*_M that starts with $\mathcal{V} = \emptyset$, and maintains the transition outputs separately. In particular, since the elements of $\mathcal{U}$ represent states in the hypothesis, a transition output has to be determined for each $u \in \mathcal{U}, a \in \Sigma$ through the query $\lambda(u,a)$. The results of these queries are stored in the corresponding rows labeled with $(\mathcal{U} \cup \mathcal{U}\Sigma) \setminus \{\varepsilon\}$. Note that $\mathcal{V} = \emptyset$ means that there can be only one equivalence class, thus the initial hypothesis is guaranteed to have a single state only.

Output Determinism

The L_1 algorithm relies on the Suffix1by1 heuristic [105, 106] for refinement, thus keeping all elements of $\mathcal{U}$ pairwisely inequivalent wrt. $\sim_\kappa$. In the general case, however, we need to strengthen

[17] Here, elements of Σ are interpreted as words of length 1.

our notion of *determinism* (cf. Definition 3.9) to also comprise *output determinism* (called "output consistency" by Van Heerdt [86]).

Definition 3.21 (*Output deterministic*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a black-box abstraction of some output function $\lambda : \Sigma^* \to \Omega^*$. $\mathcal{R}$ is called *output deterministic* if and only if

$$\forall u, u' \in \mathcal{U}, a \in \Sigma : u \sim_\kappa u' \Rightarrow \lambda(u,a) = \lambda(u',a).$$

Note that enforcing $\Sigma \subseteq Ch_\kappa(u)$ for all $u \in \Sigma^*$ trivially ensures output determinism.

If there are $u, u' \in \mathcal{U}, a \in \Sigma$ that violate output determinism, this can be resolved—in analogy to the proof of Lemma 3.5—by splitting the class $[u]_\kappa$, using a as the new discriminator.

Hypothesis Construction

If a black-box abstraction $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ satisfies the condition of output determinism in addition to closedness and determinism as defined in Definition 3.9, it is possible to construct a Mealy machine from it (in analogy to Definition 3.10).

Definition 3.22

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed, deterministic and output deterministic black-box abstraction for some output function $\lambda : \Sigma^* \to \Omega^*$. The Mealy machine corresponding to $\mathcal{R}$, Mealy($\mathcal{R}$), is the Mealy machine $\mathcal{H}$, where

- $Q_\mathcal{H} =_{df} \{[u]_\kappa \mid u \in \mathcal{U}\}$,
- $q_{0,\mathcal{H}} =_{df} [\varepsilon]_\kappa$,
- $\delta_\mathcal{H}([u]_\kappa, a) =_{df} [ua]_\kappa \;\; \forall u \in \mathcal{U}, a \in \Sigma$, and
- $\gamma_\mathcal{H}([u]_\kappa, a) =_{df} \lambda(u,a) \;\; \forall u \in \mathcal{U}, a \in \Sigma$.

Thanks to the definition of $\gamma_\mathcal{H}$, hypotheses satisfy the following modified version of invariant (I2) from Lemma 3.4 (where $\mathcal{A}$ denotes the canonical Mealy machine for λ):

(I2-Mealy) The transition outputs of a state in $\mathcal{H}$ corresponding to a prefix in $\mathcal{U}$ are correct:

$$\forall u \in \mathcal{U}, a \in \Sigma : \gamma_\mathcal{H}([u]_\kappa, a) = \gamma_\mathcal{A}(\mathcal{A}[u], a).$$

Transition Output Inconsistencies

A central result of Section 3.3 was that counterexamples constitute both reachability and output inconsistencies. However, if the initial black-box abstraction has only one equivalence class (e.g., if $\mathcal{V} = \emptyset$), reachability inconsistencies cannot occur. Luckily (as it would otherwise break the symmetry), this does not mean that the first counterexample necessarily needs to be considered as an output inconsistency. Instead, we can resolve this by supplementing the notion of reachability inconsistencies with an additional, yet very similar concept.

Definition 3.23 (*Transition output inconsistency*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic black-box abstraction of some output function $\lambda : \Sigma^* \to \Omega^*$, and let $\mathcal{H} = \text{Mealy}(\mathcal{R})$ be the corresponding hypothesis. A pair $(u,a) \in \Sigma^* \times \Sigma$ constitutes a *transition output inconsistency* if and only if $\gamma_{\mathcal{H}}(\mathcal{H}[u], a) \neq \lambda(u,a)$. (u,a) is called a *proper* transition output inconsistency if furthermore $\mathcal{H}[u] = [u]_\kappa$.

If $(u,a) \in \Sigma^* \times \Sigma$ constitutes a transition output inconsistency that is not a proper one, u constitutes a reachability inconsistency. A proper transition output inconsistency (u,a) is still very similar to a reachability inconsistency: it exposes that the representatives of $\mathcal{H}[u]$ behave differently (wrt. a) than the state reached by u in the target Mealy machine $\mathcal{M}$. Consequently, adding u to the set of short prefixes $\mathcal{U}$ will result in a violation of output determinism according to Definition 3.21.

Note that if Σ is a subset of all characterizing sets, there cannot be any proper transition output inconsistencies, as then $a \in Seps_\kappa(\mathcal{H}[u], [u]_\kappa)$.

3.5.2. Handling Counterexamples and Inconsistencies

In Section 3.3, we have shown that counterexamples can be regarded as (reachability or output) inconsistencies, and that the usual counterexample analysis techniques are more general, as they can be used to analyze these inconsistencies to achieve refinement. These results translate to Mealy machines, but some modifications are required to account for their characteristics.

The general notion of a counterexample (wrt. an hypothesis $\mathcal{H}$) is that it is a word $w \in \Sigma^*$ satisfying $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$. To make the proofs in this section work, we impose the additional assumption $\lambda_{\mathcal{H}}(w)_{|w|} \neq \lambda(w)_{|w|}$, i.e., the outputs differ in their last symbol. We do not lose generality, as every counterexample $w \in \Sigma^*$ in the general sense contains a prefix $w' \sqsubseteq_{pref} w$ satisfying the additional assumption. Moreover, assuming that $\lambda(w)$ is known, this prefix can be determined without additional membership queries. Finally, since the length of a counterexample affects the performance, it is in any case reasonable to *shorten* a counterexample as much as possible, i.e., using $w_{1..\ell}$ as the counterexample, where $\ell = \min\{i \in \mathbb{N} \mid 1 \leq i \leq |w| \wedge \lambda_{\mathcal{H}}(w)_i \neq \lambda(w)_i\}$ (resulting in the *shortest* prefix $w' \sqsubseteq_{pref} w$ that is still a counterexample). We call a counterexample obtained in this fashion a *reduced* counterexample.

Prefix-based Analysis

If $w \in \Sigma^*$ is a reduced counterexample, it is obvious that $(w_{1..|w|-1}, w_{|w|})$ constitutes a transition output inconsistency. If it is a proper one, $w_{1..|w|-1}$ is a prefix that, when added to $\mathcal{U}$, violates output determinism (which can be resolved by splitting its class using $w_{|w|}$ as discriminator). Otherwise, $w_{1..|w|-1}$ constitutes a reachability inconsistency that can be analyzed using the method described in Section 3.3.3.

Suffix-based Analysis

While, for a (reduced) counterexample $w \in \Sigma^*$, (ε, w) constitutes an output inconsistency in the usual sense, Lemma 3.8 and its proof are not applicable "as-is" in the context of Mealy machines. The reason for this is that, for an arbitrary decomposition $w = x\,y\,z$, $x,y,z \in \Sigma^*$, we generally have $\lambda(x\,y,z) \neq \lambda(x,y\,z)$ (unless $y = \varepsilon$), but merely $\lambda(x\,y,z) \sqsubseteq_{suff} \lambda(x,y\,z)$, or, alternatively, $\lambda(x\,y,z) = \lambda(x,y\,z)_{|y|+1..|y|+|z|}$ (note that the conditions in Theorem 3.3 (ii) and Lemma 3.8 (ii) are

formulated in such a way that the suffix argument to the output function is of the same length on both sides of the inequation).

Let us therefore describe how the abstract counterexample derivation has to be adapted to address this. Since the output domain is now Ω^* instead of $\mathbb{B}$ in the DFA case, the apparent choice for the effect domain is $2^{\Omega^*} \setminus \{\emptyset\}$. Observe that any output inconsistency $(x,y) \in \mathcal{U} \times \Sigma^*$ must satisfy $|y| \geq 2$, as, for all $a \in \Sigma$, $\lambda_{\mathcal{H}}^{[x]_\kappa}(a) = \gamma_{\mathcal{H}}([x]_\kappa, a) = \lambda(x,a)$ holds by construction. Furthermore, we assume that an output inconsistency (x,y) being analyzed is *reduced*, i.e., for every strict prefix $y' \sqsubset_{pref} y$, (x,y') does not constitute an output inconsistency (this apparently holds for the output inconsistency directly derived from a reduced counterexample, and can otherwise be ensured by truncating y after the first mismatch between $\lambda_{\mathcal{H}}^{[x]_\kappa}(y)$ and $\lambda(x,y)$).

To address the above-mentioned aspect that the length of the value of $\lambda(\cdot,\cdot)$ is determined by the length of the suffix argument (with the effect that $\eta(i) \cap \eta(j) = \emptyset$ for $i \neq j$), we need to re-define the effect relation. Let $\sqsubseteq \subseteq (2^{\Omega^*} \setminus \{\emptyset\}) \times (2^{\Omega^*} \setminus \{\emptyset\})$ be the relation defined via

$$X \sqsubseteq Y \Leftrightarrow_{df} \forall z \in X : \exists z' \in Y : z' \sqsubseteq_{suff} z \quad \forall \emptyset \neq X, Y \subseteq \Omega^*.$$ [18]

We will now sketch how choosing $\sqsubseteq$ as the effect relation ensures validity, and the correspondence between the breakpoint condition to the condition stated in Lemma 3.8 (ii).

Note that, for a reduced output inconsistency $(x,y) \in \mathcal{U} \times \Sigma^*$, $\eta(|v|) = \{\varepsilon\}$, thus $\eta(0) \sqsubseteq \eta(|y|)$, violating validity. However, $\eta(|y|-1) = \left\{\lambda_{\mathcal{H}}^{[x]_\kappa}(y)_{|y|}\right\}$. Since $\lambda(x,y) \in \eta(0)$, $\eta(0) \not\sqsubseteq \eta(|y|-1)$ due to (x,y) being *reduced*. Thus, the length of the abstract counterexample needs to be $|y|-1$, instead of $|y|$ in the DFA case.

Finally, let us take a look at the breakpoint condition. $\eta(i) \not\sqsubseteq \eta(i+1)$ means that there exists $u \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}))$ such that for all $u' \in \rho_{\mathcal{R}}(\delta_{\mathcal{H}}([x]_\kappa, \widehat{u}\widehat{a}))$, $\lambda(u', \widehat{v}) \not\sqsubseteq_{suff} \lambda(u, \widehat{a}\widehat{v})$, with $\widehat{u}, \widehat{a}, \widehat{v}$ as defined in Section 3.3.4. This implies $\lambda(u\widehat{a}, \widehat{v}) \neq \lambda(u', \widehat{v})$, resulting in the condition of Lemma 3.8 (ii).

3.5.3. Data Structures

A final aspect we want to discuss is how data structures change due to the adapted notion of black-box classifiers and abstractions defined in this section. As in Section 3.4.1, we will only consider observation tables and discrimination trees.

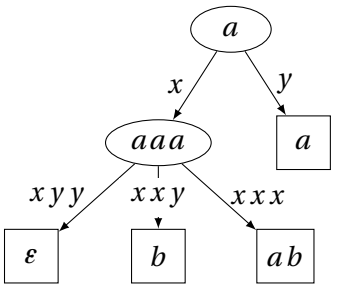
Examples of these data structures (containing output values corresponding to the Mealy machine shown in Figure 2.2b) can be seen in Figure 3.4. The observation table, shown in Figure 3.4a, contains in its cells output words of the length of the respective column header (suffix from $\mathcal{V}$), that are the suffix output with respect to the prefix labeling the row. Additionally, each row that is not labeled with ε contains the output symbol corresponding to the transition it identifies.

When looking at the discrimination tree in Figure 3.4b, one immediately notices that it is not a binary tree. An inner node labeled with $v \in \Sigma^*$ no longer has only two children (a 0- and a 1-child), but may have a child for each element in $\Omega^{|v|}$. The edge pointing to each child is labeled

[18] The correspondence between this relation and the subset relation used in the DFA case becomes apparent from the fact that changing "$\sqsubseteq_{suff}$" to "=" results in a definition of the subset relation itself. Furthermore, an alternative and possibly more efficient approach to checking whether $X \sqsubseteq Y$ holds is based on the observation that all elements of Y are of the same length ℓ. Transforming X into the set X' by replacing every element $z \in X$ with $z_{|z|-\ell+1..|z|}$ allows to reduce the test for $X \sqsubseteq Y$ to testing $X' \subseteq Y$, which can be realized more efficiently in practice by using a hash data structure.

			a	aaa
ε			x	xyy
a	/	x	y	yyy
b	/	x	x	xxy
ab	/	x	x	xxx
aa	/	y	y	yyy
ba	/	x	x	xyy
$\vdots$				

(a) Observation table



(b) Discrimination tree

Figure 3.4.: Example (Mealy) observation table and discrimination tree for the Mealy machine from Figure 2.2b

with the respective output word. A consequence of this is that, whenever some word $u \in \Sigma^*$ is sifted into the tree, the result might be a "virtual" leaf, i.e., a leaf that has a defined parent and incoming edge label, but is not part of the tree. This phenomenon, however, is not new: discovering new classes *en passant* occurs all the time when an observation table is used, and can be handled in the same way (e.g., by adding an element of $U\Sigma \setminus U$ to U if this occurs during hypothesis construction).

3.6. Discussion

We have presented a mathematical framework for formulating active automata learning algorithms following the approach of approximating the Nerode congruence by refinement. We have shown that a large number of learning algorithms, specifically those that can be regarded as variants or offsprings of the L* algorithm, can be formulated in this framework independently of their data structures. This unified description provides deep insights into the role of syntactical properties (such as prefix- or suffix-closedness), allows a precise identification of sometimes occurring deficiencies in the model and how they can be remedied, and exposes the duality of the two prevalent counterexample analysis approaches.

A similar attempt to unify the description of learning algorithms has been made by Balcázar *et al.* [25]. However, their *observation packs* framework can be considered as more of an attempt to formulate an efficient learning algorithm rather than an actual unified description of existing learning algorithms (e.g., the problem that in the original L* algorithm there can be more than one short prefix per equivalence class is simply eliminated by pointing out that, in the case of a closed and deterministic observation table, the "superfluous" short prefixes can safely be discarded). Also, the described counterexample analysis is limited to the binary search method proposed by Rivest and Schapire [155], and thus provides no further insights as to why the original one proposed by Angluin [19], or the similar one by Kearns and Vazirani [115] work.

3.6.1. Consistency Properties

An important contribution of the framework is the precise identification of the two desirable *consistency properties*, namely *reachability consistency* (Definition 3.11) and *output consistency*

(Definition 3.12), along with the syntactic properties that guarantee them (prefix-closedness of $\mathcal{U}$ and semantic suffix-closedness of the characterizing sets). Both consistency properties are crucial, as their violation means that the inferred hypothesis does not properly reflect all observations, which should however be a minimum requirement to be fulfilled by a learning algorithm.

Potential violations of these properties have long been neglected, as prefix-closedness of $\mathcal{U}$ and suffix-closedness of the global suffix set $\mathcal{V}$ of the observation table data structure were treated as properties to be enforced, at the cost of an unnecessary large number of prefixes and/or suffixes. For example, Angluin [19] proposed adding all prefixes of a counterexample as rows of the table to maintain prefix-closedness, and Maler and Pnueli [127] propose the dual strategy of adding all suffixes of a counterexample as columns of the table. Shahbaz and Groz [161] and Irfan *et al.* [106] propose several heuristics, all of which maintain suffix-closedness of the suffix set. This seems surprising, given that Rivest and Schapire [155] already showed that adding a single suffix was sufficient, and that the learning algorithm converges with a correct final hypothesis. The most likely explanation for this is that their approach was generally poorly understood, which is supported by the fact that Irfan [105] in Section 5.2.5 of his PhD thesis incorrectly claims that Rivest and Schapire's approach would result in an infinite loop.

Reachability inconsistencies can probably be considered a rather exotic phenomenon, as the only algorithm that forgoes the prefix-closedness of the short prefix set $\mathcal{U}$ is the one by Kearns and Vazirani [115]. While the consequence that this may cause a counterexample to be classified incorrectly after a single step of counterexample analysis has been observed (e.g., by Balle [26], and in fact this even occurs in the example given by Kearns and Vazirani), the usual treatment was to simply re-analyze the counterexample until it ceases to be one, instead of analyzing the reachability inconsistency as a phenomenon by its own right, as discussed in Section 3.3.3.

Lee and Yannakakis [123] observed that removing the suffix-closedness requirement of the suffix set in the observation table, as proposed by Rivest and Schapire [155], leads to non-canonical hypotheses. Steffen *et al.* [167] suggested a relaxed notion of suffix-closedness, called "semantic suffix-closedness", as a weaker property that would still ensure canonical hypotheses. Their definition of this concept however is not sufficient for ensuring canonicity and/or output consistency (in fact, our notion of *semantic suffix closedness* according to Definition 3.13 coincides with ordinary suffix-closedness for observation tables). Besides pointing out the insufficiency of the aforementioned property, Van Heerdt [86] remarked that in the case of non-canonical hypotheses, the observations stored in the table conflict with the corresponding hypothesis. He suggests to use these conflicts as counterexamples, which, when done exhaustively, will eventually guarantee canonical hypotheses. Again, this approach transforms output inconsistencies to counterexamples, without adequately addressing the nature of output inconsistencies as discussed in Section 3.3.4.

In general, there is a fascinating symmetry between the role of prefixes and suffixes in a learning algorithm. This is reflected in how the (syntactical) closedness properties ensure the (semantic) consistency properties, as discussed above, and furthermore in the counterexample analysis approaches: in prefix-based counterexample analysis according to Theorem 3.3 (i), adding the identified prefix $\widehat{u}$ to $\mathcal{U}$ in general violates prefix-closedness, while in suffix-based counterexample analysis (Theorem 3.3 (ii)), suffix-closedness is violated when a class is split using the suffix $\widehat{v}$. The inconsistencies potentially caused by violations can however be addressed using exactly the same analysis strategy that caused them in the first place, without having to analyze the full counterexample they induce.

3.6.2. Limitations

The developed framework is inherently limited to an active automata learning approach based on approximating the Nerode congruence by means of refinement. While the majority of existing learning algorithms fall into this category, it should not go unmentioned that other approaches exist.

Meinke [132] has introduced a learning algorithm for Mealy machines, called CGE (*congruence generator extension*), that takes a conceptually dual approach: instead of starting with a maximally coarse approximating relation that is refined throughout the learning process, CGE starts with a maximally fine relation (i.e., the identity relation), and coarsens it by merging equivalence classes. This corresponds to learning the *loop structure* (i.e., when are two words equivalent?) of the target automaton, whereas the framework developed in this chapter is geared towards learning algorithms that infer the *separators* of states in the target automaton. Some other algorithms following that approach, such as the IKL algorithm by Meinke *et al.* [136], have been proposed as well. Although Meinke and Sindhu [134] report superior performance for the application of *learning-based testing*, these algorithms have exponential worst-case query complexities, making them less attractive from a theoretical perspective when compared to L*-style algorithms with their polynomial query complexities.

Another category of algorithms which cannot be formulated in our framework are those that do not learn canonical DFAs (or Mealy machines). Perhaps the most prominent example is the NL* algorithm by Bollig *et al.* [36], which learns *residual finite-state automata* (RSFAs), as introduced by Denis *et al.* [62]. RSFAs are a kind of NFAs that admit a canonical form, but due to allowing non-determinism can be exponentially more succinct than their equivalent canonical DFA. Another potentially non-deterministic class of automata admitting a canonical form are universal automata [83, 124], the inference of which has been described by Björklund *et al.* [35]. In both cases, states in the hypothesis no longer necessarily correspond to the identified equivalence classes of $\sim_\kappa$. While it is certainly possible to obtain a description of the mechanisms behind the respective algorithms building upon the concepts and notation developed in this chapter, it particularly remains unsure how (or if) the counterexample analysis techniques could be translated to the case of non-deterministic hypotheses.

Finally, active learning approaches that go beyond the MAT framework of membership and equivalence queries are not covered. In particular, there are practically relevant application scenarios where queries that go beyond the power of membership queries are available: when learning *reactive systems*, the output corresponding to each input symbol can typically be observed immediately. This allows a learner to choose the next input *symbol* depending on the outputs observed so far. Thus, a membership query does not consist of a single (static) word, but is instead decomposed into an initial *reset query* and several *symbol queries*. This idea has been exploited in the field of state-machine testing by Lee and Yannakakis [122], who give an algorithm for calculating so-called *adaptive distinguishing sequences* (a clarification of the algorithm was later given by Krichen [118]). Smeenk *et al.* [164] apply this algorithm in an automata learning context as a means of approximating equivalence queries, and report that they found it to be the only viable way of finding counterexamples for large, realistic systems. The approach of using adaptive sequences directly in the learning process (and not only for conformance testing) is currently being investigated in a Master's thesis under the author's supervision [68]. First results look promising, but the technique is still preliminary and outside the scope of this thesis.

4. Discrimination Trees

In the previous chapter, we introduced the concept of black-box abstractions, which approximate the Nerode congruence $\cong_\lambda$ by means of a black-box classifier. In the abstract sense, a black-box classifier κ maps a word to a partial function from Σ^* to $\mathbb{B}$. In Section 3.4.1, two different data structures were presented for realizing such a classifier: *observation tables* and *discrimination trees*. The former uses a global set of suffixes, while for the latter the set of suffixes in the characterizing set depends on the resulting class itself.

While being slightly more difficult to implement than observation tables, discrimination trees have certain characteristics that make them the preferable data structure: two classes of its induced classifier κ always have a *unique* separator, i.e., for all $u, u' \in \Sigma^*$, $Seps_\kappa(u, u')$ is either a singleton (the element of which we will refer to by $sep_\kappa(u, u')$), or the empty set. Conversely, for every discriminator there exist at least two classes which it separates.[1] Thus, the classifier induced by a discrimination tree is *minimal* or *redundancy-free* in the sense that no inner node can be omitted without reducing its discriminatory power. Therefore, using a discrimination tree-like data structure is the conceptual counterpart of maintaining unique representatives in $\mathcal{U}$: the latter ensures determinism of the black-box abstraction, while using a discrimination tree (generally) ensures closedness.

Furthermore, as we have already pointed out in Section 3.4.1, discrimination trees allow realizing the *splitting* of classes (by means of splitting leaves) precisely as described in Definition 3.16 (p. 35), instead of a split resulting in an even more refined classifier. As refining a class requires conducting further membership queries for the elements of $\mathcal{U} \cup \mathcal{U}\Sigma$ it contains, the aforementioned property is essential for devising a learning algorithm that only poses those queries that are necessary to address the phenomena identified at an abstract level.

The above clearly motivates basing an active automata learning algorithms on the discrimination tree data structure. Kearns and Vazirani [115] presented the first algorithm that followed this approach. Since the TTT algorithm, which we will present in the next chapter, is inherently based on using a discrimination tree, we dedicate this entire chapter to the details of this data structure.

To allow a clearer focus, we will first introduce discrimination trees as a data structure in the *white-box* case, i.e., representing information about an automaton with a known and fully visible structure. We will then continue to detail on their use in a black-box (learning) setting, by presenting the Observation Pack algorithm due to Howar [93]. This algorithm also serves as the basis for the description of TTT, which then follows in the next chapter.

[1] We assume that a discrimination tree is always maintained in such a way that (i) every inner node has at least two children (resulting in the requirement of a *full* binary tree in the DFA case), and (ii) every leaf corresponds to a non-empty equivalence class (with a possible exception before the first counterexample is fully processed, as will be discussed in Section 4.2.3).

4.1. White-Box Setting

As mentioned above, we first investigate the use of discrimination trees in a *white-box* setting. This means that we consider them as a data structure to store information about a DFA $\mathcal{A}$, the structure of which is fully known, as this greatly eases reflecting on their properties and potential.

4.1.1. Formal Definitions and Notation

We start by giving a (somewhat, as will be motivated in Remark 4.1 below) formal definition of discrimination trees, complementing the intuitive presentation from Section 3.4.1.

Definition 4.1 (*Discrimination tree*)

Let Σ be an input alphabet. A $\mathbb{B}$-valued *discrimination tree* (over Σ) is a rooted directed binary tree $\mathcal{T}$, where

- the set of *nodes* is denoted by $\mathcal{N}_\mathcal{T}$, and can be written as the disjoint union of the set of *inner nodes* $\mathcal{I}_\mathcal{T}$ and the set of *leaves* $\mathcal{L}_\mathcal{T}$, i.e., $\mathcal{N}_\mathcal{T} = \mathcal{I}_\mathcal{T} \cup \mathcal{L}_\mathcal{T}$,
- the designated *root node* is denoted by $r_\mathcal{T} \in \mathcal{N}_\mathcal{T}$,
- each inner node is labeled with a *discriminator* $v \in \Sigma^*$, referred to via $n.discriminator$ for all $n \in \mathcal{I}_\mathcal{T}$,
- each inner node has exactly two children, a 0-child and a 1-child. For $n \in \mathcal{I}_\mathcal{T}$ and $o \in \mathbb{B}$, the o-child is referred to via $n.children[o]$.

The subtree rooted at the o-child of a node $n \in \mathcal{N}_\mathcal{T}$ is also referred to as the *o-subtree* of n. A node $n \in \mathcal{N}_\mathcal{T}$ is called *a child* of a node $n' \in \mathcal{N}_\mathcal{T}$ if there exists $o \in \mathbb{B}$ such that n is the o-child of n'. $n \in \mathcal{N}_\mathcal{T}$ is called a *descendant* of $n' \in \mathcal{N}_\mathcal{T}$ if there exists a sequence $n_0, \ldots, n_m \in \mathcal{N}_\mathcal{T}$, $m \in \mathbb{N}$, such that $n_0 = n'$, $n_m = n$ and, for all $0 \leq i < m$, n_{i+1} is a child of n_i. If $n \in \mathcal{N}_\mathcal{T}$ is a descendant of $n' \in \mathcal{N}_\mathcal{T}$, n' is called an *ancestor* of n. If furthermore $n \neq n'$, n is called a *proper descendant* of n', and n' is a *proper ancestor* of n. The sets of all descendants and ancestors of a node $n \in \mathcal{N}_\mathcal{T}$ is denoted by $Desc_\mathcal{T}(n)$ and $Ancest_\mathcal{T}(n)$, respectively.

Definition 4.2 (*Characterizing set, signature*)

Let $\mathcal{T}$ be a $\mathbb{B}$-valued discrimination tree over some input alphabet Σ.

- The *characterizing set* of a node $n \in \mathcal{N}_\mathcal{T}$, $Ch_\mathcal{T}(n)$, is the set of all discriminators of the proper ancestors of n, i.e.,

$$Ch_\mathcal{T}(n) =_{df} \left\{ n'.discriminator \mid n' \in Ancest_\mathcal{T}(n) \setminus \{n\} \right\}.$$

- The *signature* of a node $n \in \mathcal{N}_\mathcal{T}$, $Sig_\mathcal{T}(n)$, is defined as the set of all pairs $(v, o) \in \Sigma^* \times \mathbb{B}$ such that v is the discriminator labeling a proper ancestor n' of n, and n is in the o-subtree of n', formally:

$$Sig_\mathcal{T}(n) =_{df} \left\{ (n'.discriminator, o) \,\middle|\, n' \in Ancest_\mathcal{T}(n) \setminus \{n\} \wedge o \in \mathbb{B} \wedge n \in Desc_\mathcal{T}(n'.children[o]) \right\}.$$

Algorithm 4.1 Sifting operation in a discrimination tree $\mathcal{T}$

Require: A start node $n \in \mathcal{N}_\mathcal{T}$, an evaluation function $e : \Sigma^* \to \mathbb{B}$
Ensure: Leaf forming the result of sifting (wrt. e) being returned

```
1: function sift_T(n, e)
2:     while n ∈ I_T do                          ▷ n is not a leaf
3:         o ← e(n.discriminator)
4:         n ← n.children[o]
5:     end while
6:     return n
7: end function
```

Note that the characterizing set is the set obtained from the signature when applying the first projection on all of its elements. Alternatively, the signature can be regarded as a partial function $\Sigma^* \rightharpoonup \mathbb{B}$, and hence $Ch_\mathcal{T}(n) = \operatorname{dom} Sig_\mathcal{T}(n)$ (cf. also Definition 3.6, p. 28). The root node $r_\mathcal{T}$, finally, is the unique node in $\mathcal{T}$ satisfying $Ch_\mathcal{T}(r_\mathcal{T}) = Sig_\mathcal{T}(r_\mathcal{T}) = \emptyset$.

Visualization. An example for the visualization of a discrimination tree has already been given in Figure 3.3a: inner nodes are drawn as ellipses, while leaves are drawn as rectangles. The discriminator of an inner node constitutes its label. Edges point from an inner node to its children, where the edge to the 0-child is drawn as a dashed line, and the edge to the 1-child as a solid line. The root, finally, is the unique inner node that has no incoming edges.

Remark 4.1

Starting with this chapter, we adapt a less mathematical and more computer science-like notation, for the sake of readability. This is motivated by the fact that we will present many algorithms, for which the "dot notation" known from most object-oriented programming languages seems a more natural choice than introducing various function symbols, that are furthermore hard to memorize for the reader. Using this notation also in mathematical definition and proofs may be unconventional, but a uniform notation certainly makes for an easier understanding.

4.1.2. General Operations

Let us now introduce two important operations that can be used to obtain information from a discrimination tree: *sifting* and computing the *lowest common ancestor* (LCA). The former uses the discrimination tree for *classification*, whereas the latter emphasizes the *separation* of classes represented in a discrimination tree.

Sifting

The important operation of *sifting* has already been introduced in Section 3.4.1, in the context of active learning. Here, we will present a generalized notion that is applicable in both white-box and black-box scenarios.

Formally, we can define sifting as a higher-order function $sift_\mathcal{T} : \mathcal{N}_\mathcal{T} \times \mathbb{B}^{\Sigma^*} \to \mathcal{L}_\mathcal{T}$, which maps a *start node* $n \in \mathcal{N}_\mathcal{T}$ and an *evaluation function* $e : \Sigma^* \to \mathbb{B}$ to the leaf which forms the end result of the following process: starting with the start node $n \in \mathcal{N}_\mathcal{T}$, we first check if it is a leaf,

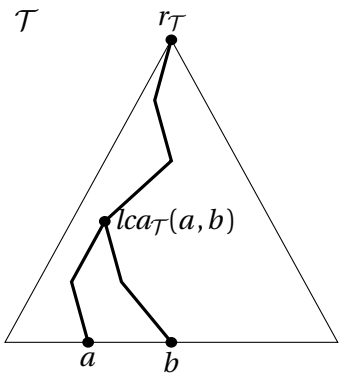


Figure 4.1.: Visualization of the role of the lowest common ancestor (LCA) in a tree

in which case we are done; the leaf then forms the result of sifting. Otherwise, if it is an inner node, we apply the evaluation function to its discriminator, i.e., compute the outcome $o =_{df} e(n.discriminator)$. This outcome determines the successor, meaning that we move to the o-child of n and repeat the described process until termination. An algorithmic description is given as Algorithm 4.1.

We will sometimes omit the explicit specification of a start node, and implicitly assume the root to be the start node. Thus, $sift_\mathcal{T}(e) = sift_\mathcal{T}(r_\mathcal{T}, e)$.

Lowest Common Ancestor

The *lowest common ancestor* of two nodes a, b in a tree (denoted by $lca_\mathcal{T}(a,b)$) is the first common node that is encountered on the respective paths from the node to the root. The significance of the lowest common ancestor stems from the fact that, when considering the paths from the root to a and b, the lowest common ancestor is the node at which those paths diverge, as visualized in Figure 4.1. Therefore, given nodes $a, b \in \mathcal{N}_\mathcal{T}$ such that neither is an ancestor of the other, we define their *separator* as $sep_\mathcal{T}(a,b) =_{df} lca_\mathcal{T}(a,b).discriminator$.

A particularly efficient way of computing the LCA is possible if nodes store a pointer to their parent node (except for the root node, for which the value of the parent pointer is assumed to be **nil**), denoted by $n.parent$, and their *depth* in the tree, denoted by $n.depth$. Here, the root is the only node satisfying $r_\mathcal{T}.depth = 0$, and all other nodes $n \in \mathcal{N}_\mathcal{T} \setminus \{r_\mathcal{T}\}$ satisfy $n.depth = n.parent.depth + 1$. Algorithm 4.2 shows how the LCA can be computed for two nodes $a, b \in \mathcal{N}_\mathcal{T}$ under these circumstances. The total number of loop iterations is bounded by $\max\{a.depth, b.depth\} - n.depth$, where n is the lowest common ancestor of a and b.

4.1.3. Discrimination Trees and Automata

Until now, we have treated discrimination trees as a general data structure, without any semantic assumptions. In particular, we left open both the significance of the leaves (which form the codomain of the *sift* operation), as well as that of the evaluation function e.

A very natural way of linking discrimination trees and automata (DFAs) is to associate the leaves of the former with (sets of) states of the latter, and use the structure of the tree for representing information about the state output functions $\lambda^q_\mathcal{A}$ of these states. This link is established

Algorithm 4.2 Lowest common ancestor computation in a discrimination tree $\mathcal{T}$

Require: Nodes $a, b \in \mathcal{N}_\mathcal{T}$, depth values and parent pointers
Ensure: Lowest common ancestor of a and b in $\mathcal{T}$ is returned

```
1:  function lca_T(a, b)
2:      if a.depth > b.depth then
3:          tmp ← a, a ← b, b ← tmp                          ▷ swap a and b
4:      end if                                    ▷ Postcondition: a.depth ≤ b.depth
5:      while a.depth < b.depth do
6:          b ← b.parent
7:      end while                                 ▷ Postcondition: a.depth = b.depth
8:      while a ≠ b do                              ▷ Invariant: a.depth = b.depth
9:          a ← a.parent
10:         b ← b.parent
11:     end while                                              ▷ Postcondition: a = b
12:     return a
13: end function
```

by the following definition.

Definition 4.3 (*Valid discrimination tree*)

Let $\mathcal{A}$ be a DFA, and let $\mathcal{T}$ be a discrimination tree where each leaf $l \in \mathcal{L}_\mathcal{T}$ is associated with a set of states of $\mathcal{A}$, referred to via $l.states \subseteq Q_\mathcal{A}$. Then, $\mathcal{T}$ is called *valid* for $\mathcal{A}$, if and only if

(i) $\pi(\mathcal{T}) =_{df} \{l.states \mid l \in \mathcal{L}_\mathcal{T}\}$ forms a partition of $Q_\mathcal{A}$, and

(ii) $\forall l \in \mathcal{L}_\mathcal{T} : \forall (v,o) \in Sig_\mathcal{T}(l) : \forall q \in l.states : \lambda^q_\mathcal{A}(v) = o$.

An alternative interpretation of the above is that for every state $q \in Q_\mathcal{A}$, $sift_\mathcal{T}(\lambda^q_\mathcal{A})$ results in the unique leaf $l \in \mathcal{L}_\mathcal{T}$ satisfying $q \in l.states$. The partition $\pi(\mathcal{T})$ is called the *induced* partition of $\mathcal{T}$. Note that $\sim_{\pi(\mathcal{T})}$ can never strictly refine $\equiv_\mathcal{A}$, as otherwise equivalent states would be separated by one of the discriminators in $\mathcal{T}$. This calls for investigating the special case that $\sim_{\pi(\mathcal{T})} = \equiv_\mathcal{A}$.

Definition 4.4 (*(Quasi-)complete discrimination tree*)

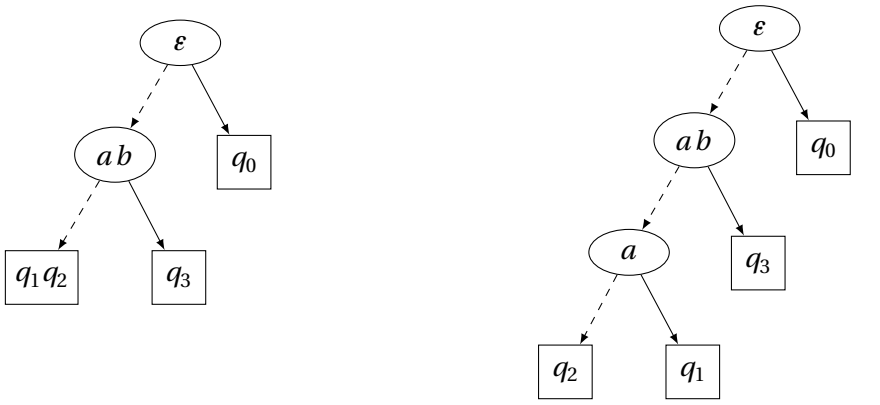
Let $\mathcal{A}$ be a DFA and let $\mathcal{T}$ be a valid discrimination tree for $\mathcal{A}$.

- $\mathcal{T}$ is called *quasi-complete* (for $\mathcal{A}$) if, for all $l \in \mathcal{L}_\mathcal{T}$, we have

$$\forall q, q' \in l.states : q \equiv_\mathcal{A} q'.$$

- $\mathcal{T}$ is called *complete* (for $\mathcal{A}$) if, for all $l \in \mathcal{L}_\mathcal{T}$, we have $|l.states| = 1$.

Note that a complete discrimination tree is also quasi-complete, but not vice versa. In both cases we have $\pi(\mathcal{T}) = Q_\mathcal{A}/\equiv_\mathcal{A}$. In the case of complete discrimination trees, this partition is furthermore the *discrete* partition of $Q_\mathcal{A}$. Thus, if $\mathcal{A}$ is canonical, every quasi-complete discrimination tree $\mathcal{T}$ for $\mathcal{A}$ is also complete, whereas for a non-canonical DFA, there cannot exist any complete discrimination trees.



(a) Valid but incomplete discrimination tree

(b) Complete discrimination tree

Figure 4.2.: Valid discrimination trees for the DFA from Figure 2.2a

Figure 4.2 shows discrimination trees for the DFA depicted in Figure 2.2a. The leaves are labeled with the sets of states associated with them. Figure 4.2a depicts a discrimination tree that is valid but incomplete, whereas Figure 4.2b depicts a complete discrimination tree for the DFA.

Separating words. We have noted above that complete discrimination trees exist for canonical DFAs only. In a canonical DFA $\mathcal{A}$, there exists a *separating word* for each pair of states $q, q' \in Q_\mathcal{A}$ ($q \neq q'$), i.e., a word $v \in \Sigma^*$ witnessing that q and q' are inequivalent due to $\lambda^q_\mathcal{A}(v) \neq \lambda^{q'}_\mathcal{A}(v)$.

A useful property of (quasi-)complete discrimination trees for a DFA $\mathcal{A}$ is that they provide a way of efficiently accessing separators for every pair of inequivalent states $q \not\equiv_\mathcal{A} q'$. Assuming that each state $q \in Q_\mathcal{A}$ stores a reference to the unique leaf $l \in \mathcal{L}_\mathcal{T}$ satisfying $q \in l.states$ (in the case of complete discrimination trees, we even have $\{q\} = l.states$), which we will refer to via $q.node$, the separator of states $q, q' \in Q_\mathcal{A}$ satisfying $q \not\equiv_\mathcal{A} q'$ can be determined as $sep_\mathcal{T}(q, q') =_{df} sep_\mathcal{T}(q.node, q'.node) = lca_\mathcal{T}(q.node, q'.node).discriminator$. For example, according to the discrimination tree from Figure 4.2b, we can determine a separator witnessing the inequivalence of q_0 and q_3 to be $sep_\mathcal{T}(q_0, q_3) = \varepsilon$.

Best and worst case depths. The *depth* of a tree $\mathcal{T}$, denoted by $depth(\mathcal{T})$, is defined as the length of the longest path from the root to a leaf (i.e., the tree consisting only of a single root leaf has depth 0). It is well known that the minimum depth of a full binary tree with n leaves is $\lceil \log n \rceil$, in which case the tree is referred to as a *balanced* binary tree, whereas the maximum depth is $n-1$. It is easy to see that there exist (families of) canonical DFAs $\mathcal{A}$ where it is possible to construct a balanced complete discrimination tree (we will give a concrete example later). The more interesting question is whether balanced complete discrimination trees can be constructed for *any* canonical DFA $\mathcal{A}$. Unfortunately, the answer is negative, as the following lemma states.

Lemma 4.1

Let Σ be an arbitrary non-empty input alphabet. There exists a family $\{\mathcal{A}_n\}_{n \in \mathbb{N}^+}$ of canonical DFAs over Σ such that $|\mathcal{A}_n| = n$, and every complete discrimination tree for $\mathcal{A}_n$ has depth $n-1$.

Proof: Fix $a \in \Sigma$ to be an arbitrary input symbol. Define $\mathcal{A}_n$ as the canonical DFA recognizing the language of all words $w \in \Sigma^*$ satisfying $\exists m \in \mathbb{N} : \#_a(w) = (n-1) + m \cdot n$ (here, $\#_a(w)$

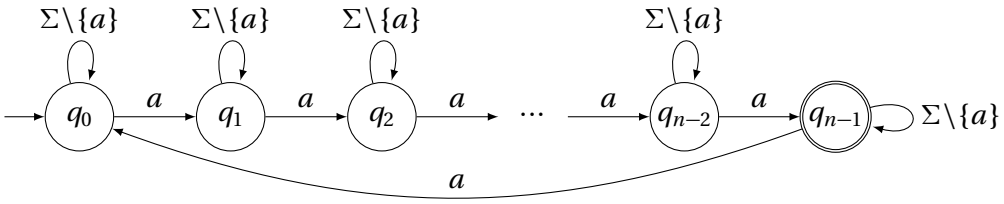


Figure 4.3.: DFA $\mathcal{A}_n$ as defined in the proof of Lemma 4.1

denotes the number of occurrences of the symbol a in w). The structure of this DFA is sketched in Figure 4.3, and it can be formally defined as:

- $Q_{\mathcal{A}_n} =_{df} \{q_i \mid 0 \leq i < n\}$,
- $q_{0,\mathcal{A}_n} =_{df} q_0$,
- $\delta_{\mathcal{A}_n}(q_i, a) =_{df} q_{(i+1) \bmod n}$ $\forall 0 \leq i < n$, $\delta_{\mathcal{A}_n}(q, a') =_{df} q$ for all $q \in Q_{\mathcal{A}}, a' \in \Sigma \setminus \{a\}$, and
- $F_{\mathcal{A}_n} =_{df} \{q_{n-1}\}$.

It is easy to see that symbols in $\Sigma \setminus \{a\}$ can never help with distinguishing the states in $Q_{\mathcal{A}_n}$. Thus, the only potential discriminators we need to consider are those of the form a^ℓ, $\ell \in \mathbb{N}$. However, for $q_i \in Q_{\mathcal{A}_n}$, $0 \leq i < n$, we have $\lambda^{q_i}_{\mathcal{A}_n}(a^\ell) = 1$ if and only if $i = n-1-(\ell \mod n)$, and $\lambda^{q_i}_{\mathcal{A}_n}(a^\ell) = 0$ otherwise. Consequently, at least one of the children of *any* inner node in any complete discrimination tree for $\mathcal{A}_n$ must be a leaf, resulting in a topology with depth $n-1$. ■

4.1.4. Computing Discrimination Trees

In the previous subsection, we have already reasoned about best- and worst-case depths of complete discrimination trees for canonical DFAs. However, we have not yet shown that there actually exists a complete discrimination trees for *every* canonical DFA. In this subsection, we will provide an algorithm to compute such a complete discrimination tree. The algorithm is similar to the famous DFA minimization algorithm due to Hopcroft [88]. In fact, the algorithm we are going to present can also be used for minimizing DFAs: when provided with a non-canonical DFA $\mathcal{A}$, it will compute a quasi-complete discrimination tree $\mathcal{T}$, i.e., the blocks of the induced partition contain equivalent states only, and thus—assuming that $\mathcal{A}$ is trim—the canonical DFA for $\mathcal{A}$ can be computed as $\mathcal{A}/{\sim_{\pi(\mathcal{T})}}$ (cf. Definition 3.1, p. 22).

The algorithm for computing a (quasi-)complete discrimination tree for a DFA $\mathcal{A}$ is given as Algorithm 4.3. As mentioned before, its structure closely resembles the minimization algorithm due to Hopcroft [88]. The main difference, apart from the discrimination tree initialization in lines 1–3 and the fact that the current partition is given implicitly by the current discrimination tree $\mathcal{T}$, is the call to the SPLIT function in line 15. Unlike in Hopcroft's original algorithm, where simply two blocks B_0, B_1 satisfying $B' = B_0 \cup B_1$ would be returned, the result of the SPLIT call is a *subtree* $\mathcal{T}'$, which is subsequently used to replace the leaf l (line 16) and thus refine the partition.

An important observation is that the algorithm can be modified to run incrementally: given a discrimination tree $\mathcal{T}$ that is valid for $\mathcal{A}$ (but not necessarily (quasi-)complete), the algorithm will augment $\mathcal{T}$ by further splitting its leaves to form a (quasi-)complete discrimination tree. This does not require any major modifications, apart from eliminating the discrimination tree

Algorithm 4.3 Compute a (quasi-)complete discrimination tree for a given DFA

Require: A DFA $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, q_{0,\mathcal{A}}, \delta_\mathcal{A}, F_\mathcal{A} \rangle$
Ensure: A (quasi-)complete discrimination tree $\mathcal{T}$ for $\mathcal{A}$

1: $\mathcal{T}_0 \leftarrow$ MAKE-LEAF$(Q_\mathcal{A} \setminus F_\mathcal{A})$ ▷ create a leaf for the state set $Q_\mathcal{A} \setminus F_\mathcal{A}$
2: $\mathcal{T}_1 \leftarrow$ MAKE-LEAF$(F_\mathcal{A})$ ▷ create a leaf for the state set $F_\mathcal{A}$
3: $\mathcal{T} \leftarrow$ MAKE-INNER$(\varepsilon, \mathcal{T}_0, \mathcal{T}_1)$ ▷ create inner node with children $\mathcal{T}_0$ and $\mathcal{T}_1$, labeled with ε
4: $W \leftarrow \emptyset$ ▷ initialize worklist
5: **if** $|F_\mathcal{A}| < |Q_\mathcal{A} \setminus F_\mathcal{A}|$ **then**
6: Add $F_\mathcal{A}$ to W
7: **else**
8: Add $Q_\mathcal{A} \setminus F_\mathcal{A}$ to W
9: **end if**
10: **while** $W \neq \emptyset$ **do**
11: $B' \leftarrow \text{poll}(W)$
12: **for** $a \in \Sigma$ **do**
13: **if** $\exists l \in \mathcal{L}_\mathcal{T} : \emptyset \neq \delta_\mathcal{A}(l.states, a) \cap B' \neq \delta_\mathcal{A}(l.states, a)$ **then** ▷ *some* transitions into B'
14: $B \leftarrow l.states$ ▷ block to be split
15: $\mathcal{T}' \leftarrow$ SPLIT$(\mathcal{A}, \mathcal{T}, B, a)$
16: REPLACE-LEAF$(\mathcal{T}, l, \mathcal{T}')$ ▷ replace leaf l in $\mathcal{T}$ with $\mathcal{T}'$
17: **if** $B \in W$ **then**
18: remove(W, B)
19: Add all newly created partition blocks to W
20: **else**
21: Add all newly created partition blocks but the largest to W
22: **end if**
23: **end if**
24: **end for**
25: **end while**
26: **return** $\mathcal{T}$

initialization performed in lines 1–3. Conversely, Algorithm 4.3 can be stopped at any time, resulting in a discrimination tree that is valid, but not necessarily quasi-complete.

There is some degree of freedom concerning how to realize the SPLIT function. In the following, we will discuss two variants. Note that both of these variants will result in a runtime complexity of at least $\Omega(kn^2)$. However, as in a learning context we are generally not too concerned with *computation* runtime (membership queries are the far more limiting factor), this should not bother us. Smetsers and Moerman [165] have recently presented an algorithm for computing a complete discrimination tree (which they call "complete splitting tree", adapting the terminology of Lee and Yannakakis [123]) in time $O(kn \log n)$. The simpler approaches presented in the following however convey the underlying ideas more clearly.

Split$_{single}$

Let us now consider the first strategy for implementing SPLIT, which we call SPLIT$_{single}$. The idea of this strategy is to consider all a-successors of the states in a block, determine the lowest

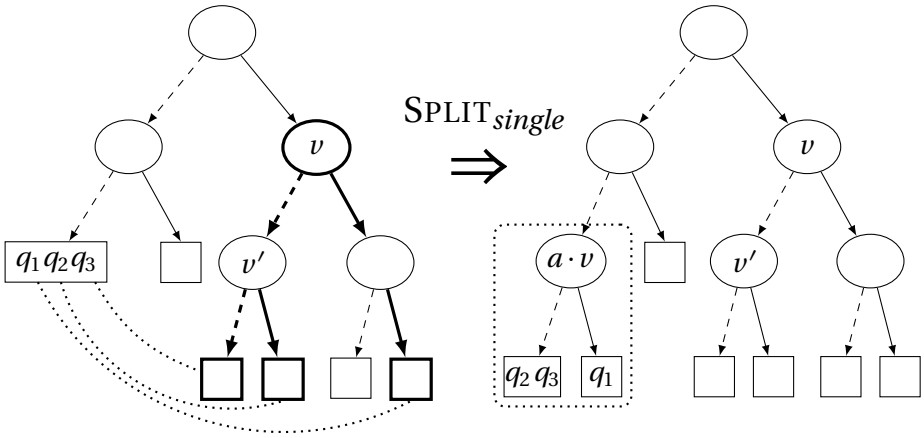


Figure 4.4.: Effect of the SPLIT$_{single}$ operation on a discrimination tree. The dotted lines correspond to the a-transitions in the automaton

common ancestor of their corresponding leaves[2] (the discriminator of which we assume to be v), and use $a \cdot v$ to split the current block. Since the choice of l (and thus B') through the **if** condition in line 13 of Algorithm 4.3 ensures that not all a-successors of states in B' point into the same block, the lowest common ancestor is guaranteed to be an inner node. Furthermore, the LCA property ensures that some of the a-transitions of states in B' point into a block in the 0-subtree of the LCA, and others point into a block in the 1-subtree. Thus, $a \cdot v$ is capable of splitting B' into two non-empty blocks.

The effect of SPLIT$_{single}$ is sketched in Figure 4.4. The dotted lines point to the blocks corresponding to the a-successors of the states q_1, q_2, q_3 of some corresponding DFA. The edges examined during the computation of the lowest common ancestor of these leaves, as well as the lowest common ancestor node itself, are drawn with bold lines.

The algorithmic realization of SPLIT$_{single}$ is given as Algorithm 4.4. The **for** loop in lines 3–11 realizes the computation of the lowest common ancestor of the nodes corresponding to the a-successor of states in B. The discriminator of this LCA is then used to obtain a new discriminator v' (line 13) that is capable of splitting B into B_0 and $B_1 = B \setminus B_0$ (line 14). These sets are then combined into a new discrimination tree $\mathcal{T}'$ with v' as its root discriminator.

Note that it is not necessary to evaluate $\lambda_{\mathcal{A}}^{q}$ for the states in B, but simply determine into which subtree of the LCA their a-transitions point. The LCA computation can even be adapted to compute the partition of states into B_0 and B_1 on-the-fly. This is however a rather technical optimization that we will not detail any further.

Split$_{tree}$

The idea of the SPLIT$_{tree}$ realization of splitting a block is to take into account the whole structure of the *subtree* rooted at the lowest common ancestor of the a-successor nodes, instead of only considering whether a transition points into its 0- and 1-subtrees. Thus, the structure induced

[2] We have introduced the concept of a lowest common ancestor for two nodes only, but it can easily be generalized to an arbitrary (finite) number of nodes. An important observation for this is that the lowest common ancestor operation is both associative and commutative.

Algorithm 4.4 SPLIT$_{single}$: split a block corresponding to a leaf in two

Require: DFA $\mathcal{A}$, discrimination tree $\mathcal{T}$, block $B \subseteq Q_{\mathcal{A}}$, symbol $a \in \Sigma$
Ensure: Discrimination tree $\mathcal{T}'$ consisting of an inner node with two leaves as children, whose state sets form a partition of B

```
1:  function SPLIT_single(A, T, B, a)
2:      succs_lca ← nil                          ▷ common LCA of a-successors of nodes in B
3:      for q ∈ B do                                              ▷ compute common LCA
4:          q' ← δ_A(q, a)
5:          n ← q'.node                  ▷ node in T corresponding to a-successor of q
6:          if succs_lca = nil then
7:              succs_lca ← n
8:          else
9:              succs_lca ← lca_T(succs_lca, n)
10:         end if
11:     end for
12:     v ← succs_lca.discriminator
13:     v' ← a · v                                  ▷ new discriminator for splitting B
14:     B_0 ← {q ∈ B | λ^q_A(v') = 0}, B_1 ← B \ B_0                ▷ partition B using v'
15:     T_0 ← MAKE-LEAF(B_0), T_1 ← MAKE-LEAF(B_1)
16:     T' ← MAKE-INNER(v', T_0, T_1)
17:     return T'
18: end function
```

by the edges that are visited during the LCA computation (i.e., the bold edges from the left of Figure 4.4) needs to be replicated by the returned tree.

Figure 4.5 gives an intuition by sketching the effect of a SPLIT$_{tree}$ operation: the original discrimination tree on the left is the same one as shown in Figure 4.4. However, instead of merely exploiting that the lowest common ancestor of the a-successors can be used to partition the block $\{q_1, q_2, q_3\}$ into $\{q_2, q_3\}$ (0-subtree) and $\{q_1\}$ (1-subtree), it is furthermore taken into account that the substructure highlighted in bold also contains the information that the node labeled with v' (now also highlighted in bold) furthermore distinguishes the a-successor blocks of q_2 and q_3. The effect can thus be described as an iterated application attempt of SPLIT$_{single}$ on all resulting non-singleton blocks, considering the same input symbol a. However, we will see that SPLIT$_{tree}$ can be realized in such a way that the subtree replacing the leaf in Figure 4.5 is "carved out" directly.

Algorithm 4.5 describes the process. For every state q in the partition block B to be split, its a-successor q' in $\mathcal{A}$ is determined. q is recorded as one of the new states corresponding to the "extracted version" of the leaf $q'.node$ via the *states* mapping (line 15). Then, n and all its ancestors are marked via the *mark* mapping (lines 8–11). The fact that MARK may only be called for a leaf, along with the upwards propagation of markings (lines 17–20) ensures that every marked inner node will have at least one marked child.

In the second phase, a new tree is *extracted* from the existing discrimination tree. Abstracting from the technical aspects that have to be addressed in the recursive implementation of the EXTRACT function, the idea can be summed up as follows:

Algorithm 4.5 SPLIT$_{tree}$: split a block by "carving out" a splitting subtree

Require: DFA $\mathcal{A}$, discrimination tree $\mathcal{T}$, block $\emptyset \neq B \subseteq Q_{\mathcal{A}}$, symbol $a \in \Sigma$

Ensure: Discrimination tree $\mathcal{T}'$ reflecting the structure of how the a-successors of states in B are split in $\mathcal{T}$

```
1:  function SPLIT_tree(A, T, B, a)
2:      for l ∈ L_T do                          ▷ Initialize mapping states from leaves to sets of states
3:          states[l] ← ∅
4:      end for
5:      for n ∈ N_T do                          ▷ Initialize mapping mark from nodes to Booleans
6:          mark[n] ← false
7:      end for
8:      for q ∈ B do                            ▷ mark all nodes corresponding to a-successors of states in B
9:          q' ← δ_A(q, a)
10:         MARK(q'.node, q)
11:     end for                                 ▷ B ≠ ∅ and upwards propagation of mark ensure mark[r_T] = true
12:     T' ← EXTRACT(r_T)
13:     return T'

14:     function MARK(l, q)                     ▷ mark a leaf l ∈ L_T and its ancestors
15:         states[l] ← states[l] ∪ {q}
16:         n ← l
17:         while n ≠ nil and ¬mark[n] do       ▷ upwards propagation of mark
18:             mark[n] ← true
19:             n ← n.parent
20:         end while
21:     end function

22:     function EXTRACT(n)                     ▷ carve out a marked subtree; precondition: mark[n] = true
23:         if n ∈ L_T then                     ▷ n is a leaf
24:             return MAKE-LEAF(states[n])
25:         else                                ▷ n is an inner node
26:             if ¬mark[n.children[0]] then    ▷ only 1-subtree is marked
27:                 return EXTRACT(n.children[1])
28:             else if ¬mark[n.children[1]] then   ▷ only 0-subtree is marked
29:                 return EXTRACT(n.children[0])
30:             else                            ▷ both subtrees are marked
31:                 T_0 ← EXTRACT(n.children[0])
32:                 T_1 ← EXTRACT(n.children[1])
33:                 v ← n.discriminator, v' ← a · v   ▷ use v' = a v as new discriminator
34:                 return MAKE-INNER(v', T_0, T_1)
35:             end if
36:         end if
37:     end function
38: end function
```

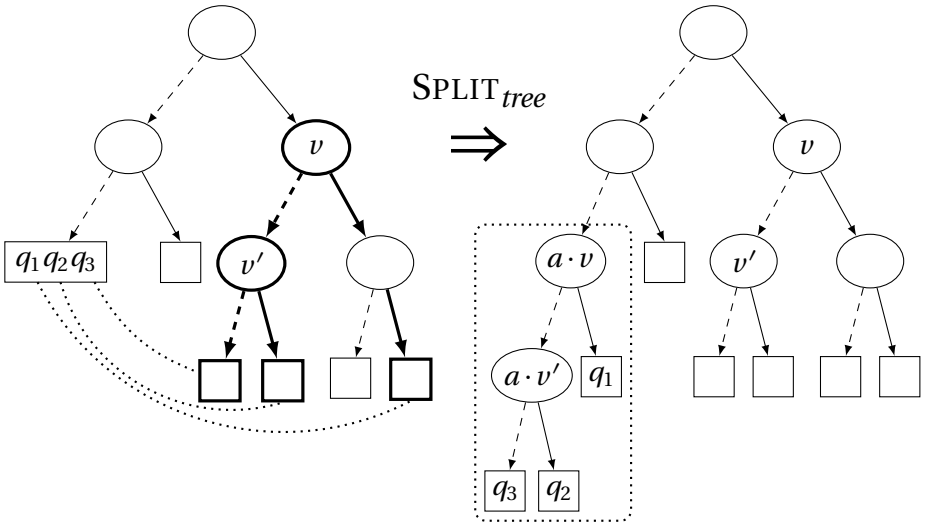


Figure 4.5.: Effect of the SPLIT_{tree} operation on a discrimination tree. The dotted lines correspond to the a-transitions in the automaton

1. Clone the original discrimination tree, replacing the blocks associated with each leaf according to the *states* mapping, and prepending a to all discriminators labeling inner nodes.

2. Remove all unmarked nodes from the discrimination tree. Since the MARK function ensures that all ancestors of a marked node are marked as well, the tree will remain connected. Furthermore, as we already observed, it is guaranteed that each inner node will have at least one child.

3. Eliminate all inner nodes with only a single child by replacing them with their child. These will be all nodes on the path from the root to the lowest common ancestor as determined by SPLIT_{single}, as well as all nodes that do not distinguish any of the a-successors (such as the inner node on the right of Figure 4.5 that is the 1-child of the node labeled with v).

Finally, the extracted discrimination tree $\mathcal{T}'$ is returned.

4.1.5. Semantic Suffix-Closedness

In Definition 3.13 (p. 31), we have already introduced *semantic suffix-closedness* in the context of black-box abstractions. This concept however can be transferred to a white-box setting as well.

Definition 4.5 (*Semantically suffix-closed discrimination trees*)

Let $\mathcal{T}$ be a valid discrimination tree for some DFA $\mathcal{A}$. For $q \in Q_{\mathcal{A}}$, let $Ch_{\mathcal{T}}(q)$ denote the characterizing set of the corresponding leaf, i.e., $Ch_{\mathcal{T}}(q) = Ch_{\mathcal{T}}(q.node)$ as defined in Definition 4.2. $\mathcal{T}$ is called *semantically suffix-closed* if and only if

$$\forall q \in Q_{\mathcal{A}}, a \in \Sigma, v \in \Sigma^* : a\,v \in Ch_{\mathcal{T}}(q) \Rightarrow v \in Ch_{\mathcal{T}}(\delta_{\mathcal{A}}(q,a)).$$

Apparently, not every valid discrimination tree is semantically suffix-closed. The question of how a semantically suffix-closed discrimination tree can be computed thus naturally arises. Luckily, this does not require a new algorithm, as the following lemma states.

Lemma 4.2

Algorithm 4.3, using $\textsc{Split}_{single}$ *(Algorithm 4.4) or* $\textsc{Split}_{tree}$ *(Algorithm 4.5) for splitting leaves, computes a semantically suffix-closed discrimination tree.*

Proof: We will only consider the case of $\textsc{Split}_{single}$, as the correctness for $\textsc{Split}_{tree}$ follows from the above observation that the latter can be regarded as an iterated application of the former.

It is easy to see that semantic suffix-closedness holds for the initially constructed discrimination tree, as the only occurring discriminator is ε. The only time the property could be violated is when new discriminators are introduced, i.e., when leaves are split by applying $\textsc{Split}_{single}$. However, the new discriminator v' is determined as $a \cdot v$ in line 13, where v is the discriminator labeling the lowest common ancestor of (the leaves corresponding to) all a-successors of states in the current block. Thus, for every $q \in B$ and $q' =_{df} \delta_{\mathcal{A}}(q,a)$, $q'.node$ is part of the subtree rooted at an inner node labeled with v (the LCA), thus preserving semantic suffix-closedness. ■

It should be noted that the above proof establishes a property that is actually stronger than mere semantic suffix-closedness: every state corresponding to a leaf that is a descendant of an inner node labeled with $a\,v$ $(a \in \Sigma, v \in \Sigma^*)$ will not only have v in the characterizing set of its a-successor, but all a-successors of all descendants will actually have the *same* ancestor node labeled with v.

The significance of semantically suffix-closed discrimination trees is further underlined by the fact that they can be stored in a very compact manner.

Proposition 4.1

A valid, semantically suffix-closed discrimination tree for a DFA $\mathcal{A}$ *with* n *states can be stored using* $\mathcal{O}(n)$ *space.*

Proof: Since no valid discrimination tree can have more than n leaves, and since every inner node has exactly two children, the overall number of nodes is bounded by $2n-1$. Furthermore, since the *states* sets of the leaves form a partition, all of those sets can be represented using in total $\mathcal{O}(n)$ space.

It remains to be shown that the discriminators do not require a superlinear amount of space. For this, first observe that semantic suffix-closedness trivially implies that the overall set of discriminators occurring in the discrimination tree is suffix-closed. It is well-known that a suffix-closed set S of size $|S| = m$ can be represented in space $\mathcal{O}(m)$ using a data structure called a *trie* (see below). As the number of inner nodes is bounded by $n-1$, there can be no more than $n-1$ distinct discriminators, hence the trie for the complete set of discriminators requires $\mathcal{O}(n)$ space. ■

In the above proof, we referred to a data structure called a *trie* [59]. Before continuing, we want to discuss this data structure in more detail. A trie is a rooted, directed tree (edges point

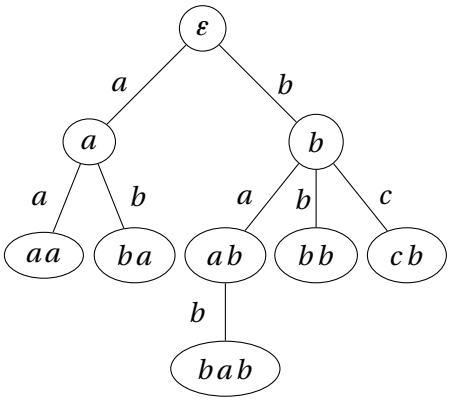


Figure 4.6.: *Trie* representing the suffix-closed set $S = \{\varepsilon, a, b, aa, ba, ab, bb, cb, bab\}$

from a node to its parent), in which every edge is labeled with a symbol from Σ. Every node corresponds to the word obtained by concatenating the symbols labeling the edges on the path from this node to the root; the root hence corresponds to the empty word ε. Note that the words corresponding to the nodes are not stored explicitly, otherwise the space complexity would be quadratic for a suffix-closed set.

An example for a trie is shown in Figure 4.6: here, the trie representing the set $S = \{\varepsilon, a, b, aa, ba, ab, bb, cb, bab\}$ is shown. Note that some nodes may have only a single child. Suffix-closedness of S ensures that for each node corresponding to a word in the set, its parent node corresponds to a word in the set, too. Thus, there are $|S|$ nodes and $|S|-1$ edges in the trie.

Let us briefly remark that it is very easy to adapt the computation of a (semantically suffix-closed) discrimination tree such that the discriminators are stored in a trie. The only time new discriminators are added (apart from the initialization in of the root with ε in line 3 of Algorithm 4.3, which can be realized by passing a reference to the root of the (otherwise empty) trie) are line 13 in Algorithm 4.4 and line 33. In both places, the new discriminator v' is of the form $v' = a\,v$. Since v is an existing discriminator that we can assume to be represented as a node in the trie, v' can be added to the trie by inserting a new node with the node corresponding to v as its parent and a as its outgoing edge label.

Semantic Suffix-Closedness and Best-Case Depth

In Lemma 4.1, we have stated that, while there might be canonical DFAs that admit complete discrimination trees of logarithmic depth, the worst-case depth of a complete discrimination tree is linear in the size of the automaton. Naturally, the additional constraint of semantic suffix-closedness cannot result in an even greater depth (as $n-1$ is the worst-case depth for any full binary tree with n leaves). However, as the following lemma states, it may result in the best-case (logarithmic) depth not being realizable.

Lemma 4.3

Given an arbitrary input alphabet Σ, there exists a family of canonical DFAs $\left\{\mathcal{A}'_n\right\}_{n\in\mathbb{N}^+}$ *such that for all $n \in \mathbb{N}^+$:*

(i) $|\mathcal{A}'_n| = n$,

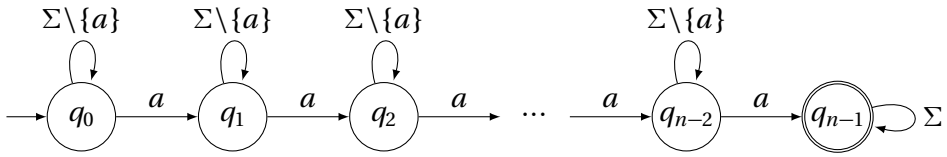


Figure 4.7.: DFA $\mathcal{A}'_n$ as defined in the proof of Lemma 4.3

(ii) there exists a complete discrimination tree of depth $\lceil \log n \rceil$ *for* $\mathcal{A}'_n$*, but*

(iii) every semantically suffix-closed complete discrimination tree for $\mathcal{A}'_n$ *has depth* $n-1$*.*

Proof: Let $n \in \mathbb{N}^+$ be a positive integer, and in the following fix an arbitrary input symbol $a \in \Sigma$. Consider the canonical DFA $\mathcal{A}'_n$ over Σ shown in Figure 4.7, accepting all words containing at least $n-1$ a's. Formally:

- $Q_{\mathcal{A}'_n} =_{df} \{q_i \mid 0 \leq i < n\}$,
- $q_{0,\mathcal{A}'_n} =_{df} q_0$,
- $\delta_{\mathcal{A}'_n}(q_i, a) =_{df} q_{\min\{i+1, n-1\}}$ for all $0 \leq i < n$, $\delta_{\mathcal{A}'_n}(q_i, a') =_{df} q_i$ for all $a' \in \Sigma \setminus \{a\}$, and
- $F_{\mathcal{A}'_n} =_{df} \{q_{n-1}\}$.

Similarly to the proof of Lemma 4.1, we do not need to consider symbols other than a as they cannot help distinguishing states (i.e., any word containing symbols other than a has the same discriminatory power after removing all those other symbols).

Let us briefly sketch the construction of a complete discrimination tree with optimal depth. Observe that any set of states $Q' = \{q_i, q_{i+1}, \ldots, q_{j-1}, q_j\}$, $i \leq j$, with contiguous indices can be partitioned into two halves of almost equal size (i.e., differing by at most one), using $a^{n-1-\lceil (i+j)/2 \rceil}$ as discriminator (resulting in $\{q_i, \ldots, q_{\lfloor (i+j-1)/2 \rfloor}\}$ and $\{q_{\lceil (i+j)/2 \rceil}, \ldots, q_j\}$). The fact that the resulting halves again consist of states with contiguous indices allows for a recursive application of this procedure, starting with $Q_{\mathcal{A}'_n}$, and obviously resulting in a discrimination tree with logarithmic depth.

We will now outline why a logarithmic depth cannot be achieved when respecting the restriction of semantic suffix-closedness. First, observe that no discriminator other than a^{n-2} can distinguish q_0 and q_1. Since the overall set of discriminators needs to be suffix-closed, this means that all $n-1$ discriminators $a^{n-2}, a^{n-3}, \ldots, a^1, a^0 = \varepsilon$ are required. Since there are exactly $n-1$ inner nodes in a complete discrimination tree for a canonical DFA of size n, the discriminators of all inner nodes must be pairwisely distinct.

Now, observe that the root discriminator must be either ε or a^{n-2}. Any other discriminator a^ℓ, $1 \leq \ell < n-2$, partitions the set $Q_{\mathcal{A}'}$ into two non-singleton sets $\{q_0, q_1, \ldots, q_{n-2-\ell}\}$ and $\{q_{n-1-\ell}, \ldots, q_{n-1}\}$. However, since $a^\ell \in Ch(q_0)$ and $a^\ell \in Ch(q_{n-2})$, and since $\delta_{\mathcal{A}'_n}(q_0, a) = q_1$ and $\delta_{\mathcal{A}'_n}(q_{n-2}, a) = q_{n-1}$, semantic suffix-closedness would require both $a^{\ell-1} \in Ch(q_1)$ and $a^{\ell-1} \in Ch(q_{n-1})$. Since q_1 and q_{n-1} are in different subtrees of the root (which is labeled with a^ℓ), this would require a node with discriminator $a^{\ell-1}$ to be present in both the 0- and the 1-subtree of the root, which is impossible as every discriminator can only occur once. Thus, the partitioning induced by the root discriminator necessarily contains a singleton block,

and the other block has contiguous indices. The argumentation can be continued recursively to show that in every step of the discrimination tree construction, semantic suffix-closedness can only be preserved with an "extreme" choice for the discriminator (i.e., one of the resulting blocks is a singleton), resulting in a topology realizing the worst-case depth of $n-1$. ■

The above lemma should however not lead to the impression that semantically suffix-closed discrimination trees are inferior. First, they generally admit a maximally compact representation, as we have shown above. Second, they can be computed by inspecting local properties only (i.e., the immediate successors of a state), whereas the construction of the tree with logarithmic depth in the above proof requires knowledge about the global structure of the automaton (a fact that is of particular importance in a learning context, as we will see in the next section). Third, computing a discrimination tree with optimal depth is generally a hard problem [104]. It nevertheless highlights a dilemma that is often encountered in active automata learning: the *principled* way of finding a solution works well in the average case, but it is almost always possible to find single instances where *heuristics* perform better.

4.2. Black-Box Setting: Learning with Discrimination Trees

In this section, we will describe a discrimination tree-based learning algorithm, commonly called Observation Pack [93, 108], which can be regarded as a precursor to the TTT algorithm that we will present in the next chapter. The Observation Pack algorithm is a straightforward adaption of Rivest and Schapire's algorithm [155], replacing the observation table data structure with a discrimination tree.

4.2.1. Discrimination Trees as Black-Box Classifiers

Discrimination trees as black-box classifiers were introduced informally in Section 3.4.1. The formalization of discrimination trees presented in the previous section calls for some brief remarks on how the concepts can be adapted to a black-box setting.

As usual, we assume that there is some suffix-observable output function $\lambda : \Sigma^* \to \mathbb{B}$ for which we want to infer a model, and that we can evaluate via membership queries. Our formal description of *sifting* in Section 4.1.2 (cf. also Algorithm 4.1) requires as argument an *evaluation function* $e : \Sigma^* \to \mathbb{B}$. Since we want to use a discrimination tree $\mathcal{T}$ to *classify* a word $u \in \Sigma^*$, we will use as evaluation function the *residual output function* of u wrt. λ, i.e., $u^{-1}\lambda$ (cf. Definition 3.3, p. 24). As we usually assume λ to be fixed, we will also simply refer to this as sifting the *prefix* u into a discrimination tree $\mathcal{T}$, i.e., $sift_{\mathcal{T}}(u) =_{df} sift_{\mathcal{T}}(u^{-1}\lambda)$ for all $u \in \Sigma^*$. A discrimination tree $\mathcal{T}$ therefore induces an equivalence relation $\sim_{\mathcal{T}} \subseteq \Sigma^* \times \Sigma^*$, defined via $u \sim_{\mathcal{T}} u' \Leftrightarrow_{df} sift_{\mathcal{T}}(u) = sift_{\mathcal{T}}(u')$. This relation is refined by $\cong_{\lambda}$, and if we ensure that ε is the root discriminator of $\mathcal{T}$, it furthermore saturates $\lambda^{-1}(1)$. Thus, a discrimination tree black-box classifier can be cast into the notion of a suffix-based black-box classifier $\kappa \in K_{\lambda}$ according to Definition 3.5 (p. 28) by defining $\kappa(u) =_{df} Sig_{\mathcal{T}}(sift_{\mathcal{T}}(u))$, where the signature $Sig_{\mathcal{T}}(sift_{\mathcal{T}}(u)) \subseteq \Sigma^* \times \mathbb{B}$ is treated as a partial function. The notion of characterizing and separator sets (cf. Definition 3.6, p. 28) translates accordingly, where the structure of a discrimination tree guarantees that $Seps_{\mathcal{T}}(u, u')$ is a singleton if $u \not\sim_{\mathcal{T}} u'$ (and the empty set otherwise), the unique element of which we will refer to

by $sep_{\mathcal{T}}(u,u')$. This motivates to identify κ and $\mathcal{T}$, (e.g., we define $Ch_{\mathcal{T}}(u) =_{df} Ch_{\mathcal{T}}(sift_{\mathcal{T}}(u))$), and, as a further simplification, we identify leaves of $\mathcal{T}$ with equivalence classes of $\sim_{\mathcal{T}}$ (i.e., $[u]_{\mathcal{T}} = sift_{\mathcal{T}}(u)$). Note that we always assume $\mathcal{T}$ to be a full binary tree, meaning that sifting is a total function (possibly resulting in the creation of *unlabeled* leaves on-the-fly).

4.2.2. Spanning-Tree Hypothesis

In many descriptions of learning algorithms, a clear distinction between the central data structure—e.g., an observation table, storing the observations plus some additional data, such as the set of short prefixes $\mathcal{U}$—and the hypothesis constituting the output of the learning algorithm can be observed. More precisely, the observation data structure is what is being built during the actual learning phase, and the hypothesis is then constructed in a separate step, from the information stored in the observation data structure. For example, Angluin [19] describes how a DFA can be built *from scratch* from a closed and "consistent" (in our terminology: *deterministic*) observation table, and other authors such as Rivest and Schapire [155] or Kearns and Vazirani [115] follow a similar pattern.

Howar *et al.* [94] have observed that it is much more adequate to use the hypothesis itself as a prime representation of (parts of) the knowledge of the learner, which grows and evolves over the entire course of the learning process, instead of repeatedly being reconstructed from scratch. This is particularly effective in the case of learning algorithms that maintain a prefix-closed set of unique representatives $\mathcal{U}$. In this case, the learner's knowledge about the structure of the target system can be maintained in a *spanning-tree hypothesis*, with the following characteristics:

(C1) The spanning-tree hypothesis grows monotonically over the course of the entire learning process, i.e., states are added, but never removed. Furthermore, whenever the target of a transition changes, the new target can only be a newly introduced state (this is justified by invariant (I3) stated in Lemma 3.4, p. 32).

(C2) Each of the k outgoing transitions of a state is either a *tree transition* or a *non-tree transition*.

(C3) An outgoing *tree transition* of a state stores a reference to its target state, and the target state of a tree transition will never change.

(C4) The set of all *tree transitions* forms a directed *spanning tree*, the root of which is the initial state. Every state other than the initial one has a unique *incoming tree transition*.

(C5) The representative prefix associated with a state is called its *access sequence*, and it can be obtained by concatenating all transition labels on the path from the root of the spanning tree to the respective state (similar to a *trie*, but in reverse direction). This ensures that representative prefixes are unique, that the access sequence of the initial state is ε, and that the set $\mathcal{U}$ of all representative prefixes is prefix-closed.

(C6) The *(transition) access sequence* of the outgoing a-transition of a state with access sequence $u \in \mathcal{U}$ is defined as ua. In the case of tree transitions, this is the same as the access sequence of the target state.

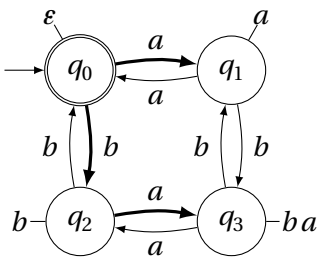


Figure 4.8.: Spanning-tree hypothesis, corresponding to the short prefix set $\mathcal{U} = \{\varepsilon, a, b, ba\}$. States are annotated with their access sequences

(C7) A *non-tree transition* with access sequence $ua \in \mathcal{U}\Sigma$ stores the characterizing observations (wrt. the employed black-box abstraction, see below) for ua.

(C8) States are added to the hypothesis by converting non-tree transitions into tree transitions, pointing to the new state.

Instead of having to create a new DFA to return as its intermediate hypothesis, a learner can simply return a *view* on the (evolving) spanning-tree hypothesis, hiding the distinction between tree and non-tree transitions, and transparently substituting the correct target states for the characterizing observations stored with the non-tree transitions (provided that the employed black-box abstraction is complete and deterministic).

An example spanning-tree hypothesis is depicted in Figure 4.8. Here, tree transitions are rendered in bold, and the states are annotated with their access sequences.

Combination with Discrimination Trees

In (C7) we have stated that a non-tree transition in the spanning-tree hypothesis stores the characterizing observations for its access sequences. If the black-box classifier is realized by means of a discrimination tree, this simply means that a non-tree transition points to a *node* in the discrimination tree, as opposed to tree transitions, which point to a state in the hypothesis.

Figure 4.9 illustrates this. As usual, tree transitions are drawn in bold. The pointer from non-tree transitions in the hypothesis (left) to nodes in the discrimination tree (right) are visualized using dotted lines. The a-transition of q_2 as well as the b-transitions of q_1 (omitted for the sake of readability) and q_2 point to leaves in the discrimination tree, which correspond to states in the hypothesis. The b-transition of q_0, on the other hand, points to an inner node of the discrimination tree. This corresponds to an uncertainty regarding the target of this transition, represented as non-determinism in the hypothesis: any of the states corresponding to leaves in the subtree rooted at this inner node are possible candidate targets. The non-determinism can be resolved by sifting the transition (rather: its access sequence b) further down the tree.[3]

4.2.3. The Observation Pack Algorithm

We will now present the Observation Pack algorithm by Howar [93, 108], which is a rather straightforward combination of the discrimination tree data structure with the counterexample

[3] Note that this manifestation of non-determinism is not to be confused with non-determinism of black-box abstractions as defined in Definition 3.9, which arises due to an insufficiently refined black-box classifier.

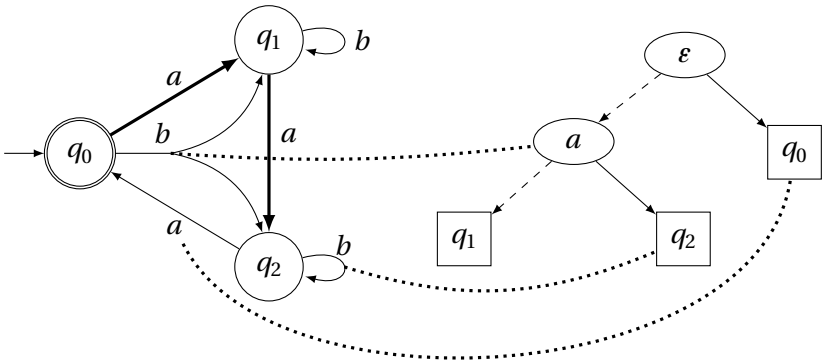


Figure 4.9.: Example illustrating the connection between a spanning-tree hypothesis (left) and its associated discrimination tree (right). The b-transition of q_0 points to an inner node in the discrimination tree and thus introduces non-determinism, while the a- and b-transitions of q_2 point to a leaf and therefore are deterministic

analysis proposed by Rivest and Schapire [155]. As observed in Section 3.2.1, a learning algorithm can conceptually be split into two operations: *initialization* and *refinement* (i.e., counterexample processing). We will describe the data structures used by Observation Pack along with the fairly trivial initialization phase, and will then take a look at how refinement is achieved. Both phases will then be illustrated along an example run. We conclude our presentation of Observation Pack with a brief complexity analysis.

Initialization and Data Structures

The initialization phase of the Observation Pack algorithm can be described very concisely: create a new single-state hypothesis and a discrimination tree consisting of an inner node (labeled with ε) and two leaves, and determine the leaf corresponding to the initial state by sifting ε into the tree. Then, determine the targets of the transitions of the initial state.

The corresponding pseudocode is given as Algorithm 4.6. The **if** block in lines 6–10 determine whether the initial state is accepting or not by looking at whether the corresponding leaf is in the 0- or the 1-subtree of the root (throughout the algorithm, the acceptance of states is always determined in this way). The call to LINK in line 11 establishes the link between a state $q \in Q_{\mathcal{H}}$ and a leaf $l \in \mathcal{L}_{\mathcal{T}}$: we assume that each state q has a pointer to its corresponding node in the discrimination tree (referred to via $q.node$), and conversely, every leaf l has a pointer to the state it corresponds to (referred to via $l.state$, which may be **nil**). Thus, LINK(l,q) establishes $l.state = q \wedge q.node = l$.

The last line before the **return** statement (line 12) refers to a function called CLOSETRANSITIONS, also shown in Algorithm 4.6. We assume that the outgoing transitions of every state $q \in Q_{\mathcal{H}}$ can be accessed as $q.trans[a]$ for $a \in \Sigma$. Furthermore, every outgoing transition t can either be a tree or a non-tree transition. As mentioned in the previous section, non-tree transitions point to nodes in the discrimination tree $\mathcal{T}$. For a non-tree transition t, this target node is referred to via $t.tgt\_node$. Every transition is initialized with $t.tgt\_node = r_{\mathcal{T}}$, i.e., pointing to the root of the discrimination tree. We furthermore assume that for a tree

Algorithm 4.6 Initialization routine for the Observation Pack algorithm

Require: Access to suffix-observable output function $\lambda : \Sigma^* \to \mathbb{B}$ (implicit)
Ensure: Initial hypothesis $\mathcal{H}$ with corresponding discrimination tree $\mathcal{T}$

function OBSERVATIONPACK-INIT
  $\mathcal{H} \leftarrow$ CREATEHYPOTHESIS() ▷ create new hypothesis with single state $q_{0,\mathcal{H}}$
  $\mathcal{T}_0 \leftarrow$ MAKE-LEAF(**nil**), $\mathcal{T}_1 \leftarrow$ MAKE-LEAF(**nil**) ▷ initialize discrimination tree
  $\mathcal{T} \leftarrow$ MAKE-INNER($\varepsilon, \mathcal{T}_0, \mathcal{T}_1$)
  $l \leftarrow sift_{\mathcal{T}}(\varepsilon)$ ▷ sift ε (representing $q_{0,\mathcal{H}}$) into the tree
  if $(\varepsilon, 1) \in Sig_{\mathcal{T}}(l)$ **then** ▷ node corresponding to $q_{0,\mathcal{H}}$ is in 1-subtree of the root
    $F_{\mathcal{H}} \leftarrow \{q_{0,\mathcal{H}}\}$
  else ▷ node corresponding to $q_{0,\mathcal{H}}$ is in 0-subtree
    $F_{\mathcal{H}} \leftarrow \emptyset$
  end if
  LINK($l, q_{0,\mathcal{H}}$) ▷ establish link between leaf and state
  CLOSETRANSITIONS($\mathcal{H}, \mathcal{T}$)
  return $\langle \mathcal{H}, \mathcal{T} \rangle$
end function

Require: (Unclosed) spanning-tree hypothesis $\mathcal{H}$, discrimination tree $\mathcal{T}$
Ensure: Hypothesis $\mathcal{H}$ is closed

procedure CLOSETRANSITIONS($\mathcal{H}, \mathcal{T}$)
  $N \leftarrow \emptyset$ ▷ transitions pointing to new states
  do
    while $Open(\mathcal{H}) \neq \emptyset$ **do**
      $t \leftarrow$ choose($Open(\mathcal{H})$)
      $tgt \leftarrow sift_{\mathcal{T}}(t.tgt\_node, t.aseq)$ ▷ sift transition further down the tree
      $t.tgt\_node \leftarrow tgt$
      if $tgt \in \mathcal{L}_{\mathcal{T}}$ **and** $tgt.state =$ **nil then** ▷ discovered new state ("unclosedness")
        $N \leftarrow N \cup \{t\}$
      end if
    end while
    if $N \neq \emptyset$ **then**
      $t \leftarrow$ choose(N) ▷ e.g., transition with minimal access sequence
      $q \leftarrow$ MAKETREE(t) ▷ convert t into a tree transition, adding a new state
      $N \leftarrow \{t' \in N \mid t'.tgt\_node \neq t.tgt\_node\}$ ▷ update N
      LINK($t.tgt\_node, q$)
    end if
  while $N \neq \emptyset$
end procedure

Algorithm 4.7 Realization of refinement in the Observation Pack algorithm

```
1:  procedure ObservationPack-Refine(H, T, w)
2:      ⟨û, â, v̂⟩ ← Analyze-OutIncons(ε, w)          ▷ suffix-based analysis
3:      Split(H, T, û, â, v̂)                          ▷ split state in H and leaf in T
4:      CloseTransitions(H, T)
5:  end procedure

6:  procedure Split(H, T, û, â, v̂)
7:      q_pred ← H[û]
8:      t ← q_pred.trans[â]
9:      q_old ← t.tgt_state
10:     q_new ← MakeTree(t)                           ▷ turn t into a tree transition
11:     ⟨l_0, l_1⟩ ← Split-Leaf(q_old.node, v̂)       ▷ replace leaf with inner node and two leaves
12:     if λ(⌊q_old⌋, v̂) = 0 then
13:         Link(l_0, q_old)
14:         Link(l_1, q_new)
15:     else
16:         Link(l_0, q_new)
17:         Link(l_1, q_old)
18:     end if
19: end procedure
```

transition t, $t.tgt\_node$ refers to the leaf associated with its target state. The target state of a (tree or non-tree) transition t is referred to via $t.tgt\_state$.

A non-tree transition whose target node is not a leaf is referred to as an *open transition* (in this case, $t.tgt\_state$ will be **nil**), and $Open(\mathcal{H})$ denotes the set of all open transitions in $\mathcal{H}$. In the body of the **while** loop (lines 19–24), a single transition $t \in Open(\mathcal{H})$ is selected and *closed*, by sifting it further down the tree until it points to a leaf (the sifting is performed using its access sequence, which we refer to via $t.aseq$). If this leaf has no associated state (i.e., $l.state = \textbf{nil}$), the transition is recorded in a set N of transitions pointing to new states. This basically constitutes an unclosedness. Note that in the case of DFA learning, there can only be one such situation in the entire course of the algorithm: the first time a state with an acceptance value different from that of the initial state is discovered.

Finally, in lines 27–30, one of the transitions pointing to an undiscovered state, say t, is selected (e.g., by choosing the transition with a shortest access sequence among all transitions in N), and then converted into a tree transition (line 28). This results in a new state being added to $\mathcal{H}$, which is subsequently linked with the target node of t. The acceptance value of this new state is again determined by considering in which of the root's subtrees its corresponding node is (omitted for the sake of brevity). The introduction of a new state results in new (open) transitions which need to be closed, thus the outer **do**..**while** loop is executed again, until no further states are added.

Refinement

The idea of how the hypothesis and discrimination tree are refined in the Observation Pack algorithm is easily explained. The corresponding pseudocode is shown as Algorithm 4.7. First, note that every state of $\mathcal{H}$ has a unique representative short prefix, namely its access sequence, which is referred to via $\lfloor q \rfloor$. This allows the simpler formulation of suffix-based counterexample analysis, as stated in Remark 3.5 (p. 43).

Theorem 3.3 (ii) (p. 37) states that a counterexample $w \in \Sigma^*$ can be decomposed into $w = \widehat{u}\widehat{a}\widehat{v}$ with the following property: let $q_{pred} =_{df} \mathcal{H}[\widehat{u}]$ and $q_{old} =_{df} \mathcal{H}[\widehat{u}\widehat{a}]$, we then have $\lambda(\lfloor q_{pred} \rfloor \widehat{a}, \widehat{v}) \neq \lambda(\lfloor q_{old} \rfloor, \widehat{v})$. Thus, the $\widehat{a}$-successor of q_{pred} must be different from q_{old}, which calls for the introduction of a new state q_{new}. This state is created by converting the $\widehat{a}$-transition of q_{pred} into a tree transition (line 10). The leaf that formerly corresponded to q_{old} is split and replaced[4] by an inner node with discriminator $\widehat{v}$ and two leaves. The states q_{old} and q_{new} are then linked to these leaves, according to their future behavior wrt. $\widehat{v}$ (lines 12–18; note that the future behavior wrt. $\widehat{v}$ has already been tested in the course of counterexample analysis, thus testing the **if** condition requires no additional membership query). Finally, the open transitions in $\mathcal{H}$ are closed (line 4). This comprises the new transitions of q_{new}, but also the non-tree transitions that used to point to q_{old}: for them, one needs to determine whether they keep pointing to q_{old} or whether their target changes to q_{new}, by testing them against $\widehat{v}$.[5]

An Example Run

Let us now briefly take a look at how Observation Pack infers a model of a concrete automaton. As the target DFA, we choose the one from Figure 2.2a, accepting words with an even number of a's and b's.

The initial state is obviously accepting. During initialization, both a and b are found to lead to an undiscovered non-accepting state, which triggers the introduction of a new state (using a as its access sequence). The corresponding spanning-tree hypothesis is shown in Figure 4.10a, and the corresponding discrimination tree in Figure 4.10b.

The initial hypothesis $\mathcal{H}$ classifies some words incorrectly. One of these words is $w = baaaaaab$, since $\lambda_{\mathcal{H}}(w) = 0$, but $\lambda(w) = 1$. Applying suffix-based counterexample analysis (cf. Section 3.3.4), a decomposition $\langle \widehat{u}, \widehat{a}, \widehat{v} \rangle = \langle \varepsilon, b, aaaaaab \rangle$ is determined.[6] Thus, the b-transition of $q_{pred} = \mathcal{H}[\varepsilon] = q_0$ is converted into a tree transition, resulting in the introduction of a new state. Furthermore, the leaf in the discrimination tree corresponding to $q_{old} = \mathcal{H}[b] = q_1$ is split, using $aaaaaab$ as the discriminator. The resulting data structures, after closing all open transitions, are shown in Figures 4.10c and 4.10d.

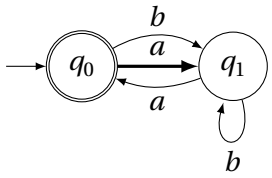
The refined hypothesis $\mathcal{H}'$ from Figure 4.10c still classifies $w = baaaaaab$ incorrectly. A second suffix-based counterexample analysis yields the decomposition $\langle \widehat{u}, \widehat{a}, \widehat{v} \rangle = \langle b, a, aaaaab \rangle$.[7]

[4] It would perhaps be more adequate to say that the leaf is *converted* to an inner node, as the inner node is meant to be the same object as the leaf. This is of importance, as otherwise the *tgt_node* pointer of the incoming non-tree transitions of q_{old} would be invalidated, instead of pointing to the inner node.

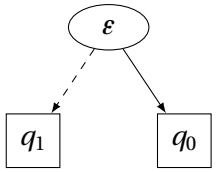
[5] It makes sense to maintain a separate, global list of open transitions instead of scanning the entire hypothesis for open transitions every time CLOSETRANSITIONS is called. This can be accomplished easily by maintaining a list of *incoming non-tree transitions* for each state. Every time the leaf corresponding to a state is split, all those transitions are added to the list of open transitions.

[6] This decomposition is in fact the only one satisfying the conditions of Theorem 3.3 (ii).

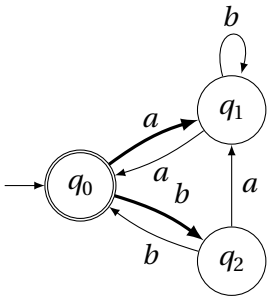
[7] Again, this is the only valid decomposition.



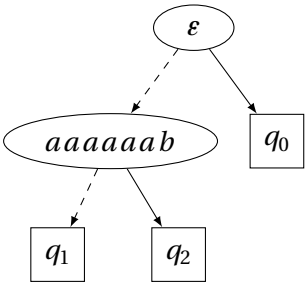
(a) Initial hypothesis



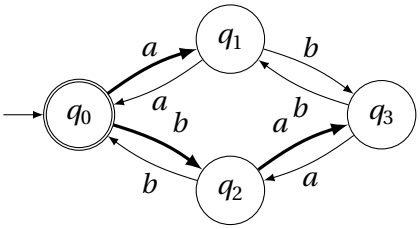
(b) Initial discrimination tree



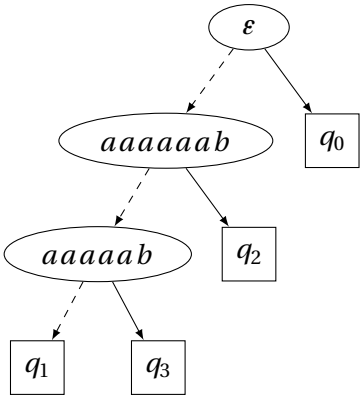
(c) Hypothesis after first refinement



(d) Discrimination tree after first refinement



(e) Hypothesis after second refinement



(f) Discrimination tree after second refinement

Figure 4.10.: Evolution of hypothesis and discrimination tree during a run of Observation Pack

Converting the a-transition of $q_{pred} = \mathcal{H}'[b] = q_2$ into a tree transition, and splitting the leaf associated with $q_{old} = \mathcal{H}'[b\,a] = q_1$ using $a\,a\,a\,a\,a\,b$ as discriminator results in the final hypothesis shown in Figure 4.10e. Figure 4.10f shows the accompanying discrimination tree.

Complexity

In the following, we analyze the worst-case complexity of the Observation Pack algorithm, using the parameters n, k, m as described in Section 3.2.1 for describing the input size, and assuming $m = \Omega(n)$.

Query Complexity. The majority of membership queries results from sifting non-tree transitions into the tree. Since this is done incrementally, no more than $n-1$ membership queries ($n-1$ being the worst-case depth of a discrimination tree) will be required per transition, resulting in a total of $\mathcal{O}(k\,n^2)$ membership queries during sifting (note that, although asymptotically irrelevant, tree transitions also factor into that number, as each tree transition is derived from a non-tree transition that had to be sifted to some level of the discrimination tree). Counterexample analysis can be done using $\mathcal{O}(\log m)$ queries per counterexample (cf. Proposition 3.3, p. 44), contributing another $\mathcal{O}(n \log m)$ membership queries. This yields an overall membership query complexity of $\mathcal{O}(k\,n^2 + n \log m)$.

Symbol Complexity. Since the set of access sequences is prefix-closed, no access sequence contains more than n symbols. Thus, all $n-1$ counterexample analysis steps combined require $\mathcal{O}(n\,m \log m)$ symbols (as already stated in Proposition 3.3). The discriminators in the discrimination tree (except for the root discriminator ε) are obtained as suffixes of provided counterexamples, thus their length can only be bounded by m. Since these suffixes are used for queries during hypothesis construction (i.e., closing transitions), the asymptotic upper bound for the symbol complexity is $\mathcal{O}(k\,n^2\,m + n\,m \log m)$.

Space complexity. The spanning-tree hypothesis can be stored in space $\Theta(k\,n)$ (note that access sequences of states do not need to be stored explicitly, as they are determined by the spanning tree). The discrimination tree has $2n-1$ nodes, and each of the $n-1$ inner nodes needs to store a discriminator of length in $\mathcal{O}(m)$. This results in an overall space complexity in $\mathcal{O}(k\,n + n\,m)$.

4.2.4. A Note on Discrimination Tree-based Learning Algorithms

All active automata learning algorithms that we have considered so far require $n-1$ equivalence queries in the worst case. Furthermore, as already mentioned in Section 3.4.3, Balcázar *et al.* [25] have shown that any algorithm with a polynomial membership query complexity requires $\Omega(n/\log n)$ equivalence queries in the worst case. In practice, however, discrimination tree-based algorithms typically require much more counterexamples than their observation table-based counterparts. Howar [93], in Section 2.2.4 of his PhD thesis, even reports that "it has often been argued that using this strategy [of splitting only a single class] for handling counterexamples is not a wise choice in practice since the number of equivalence queries increases drastically." This harsh judgment is due to the fact that, in practice, equivalence queries are often unavailable and need to be approximated using membership queries. Techniques that provide guarantees—such as correctness under the assumption that the target system has no more than Δn additional states wrt. the current hypothesis, as is the case for the W-method due

to Chow [52]—typically even require exponentially many membership queries. Thus, (approximated) equivalence queries are generally regarded as being very expensive.

However, the seemingly better equivalence query complexity of observation table-based algorithms is, once again, due to heuristics. Since filling an observation table requires posing much more membership queries than sifting access sequences into a discrimination tree, less equivalence queries are to be expected, as *every* query that is not strictly necessary can expose diverging behavior. Consider a discrimination tree-based algorithm that, after constructing a hypothesis, additionally poses membership queries for *all* combinations of elements in $\mathcal{U} \cup \mathcal{U}\Sigma$ and discriminators in the discrimination tree, regardless of where they occur. Any observed diverging outputs could then be used as counterexamples, reducing the apparent number of equivalence queries precisely at the cost of posing more membership queries.

The above justifies to consider filling an observation table as more of a built-in heuristic for approximating equivalence queries. It is however hard to argue why this should be a feature of an algorithm itself, since such heuristics can just as well be applied "on top", furthermore leaving the freedom to resort to potentially better heuristics (e.g., the evolving hypothesis approach proposed by Howar *et al.* [93, 94]). Moreover, there may be scenarios where membership queries are expensive, but inexpensive sources of counterexamples exist, such as in *black-box checking* [81, 82, 149, 150] or *learning-based testing* [134, 136, 163]. Here, model checking is used to check (intermediate) hypotheses against a specification, which may result in *spurious* counterexamples, i.e., apparent violations of the specification that are merely due to incorrect hypotheses. Exploiting such sources of counterexamples exhaustively before resorting to methods such as random testing, as proposed by Meinke *et al.* [136], helps steering the learning process in a direction relevant to the specification, and furthermore allows random (model-based) testing techniques to use a more refined model as a basis.

Undoubtedly, using a discrimination tree-based algorithm in a practical setting forces the user to put more thought into how equivalence queries can be realized or approximated. But then again, merely relying on the heuristics implicitly encoded in the observation table data structure is not a good idea to begin with, either.

5. The TTT Algorithm

In the previous chapter, we have presented the Observation Pack algorithm, which combines the discrimination tree data structure originally introduced by Kearns and Vazirani [115] with the binary search counterexample analysis strategy due to Rivest and Schapire [155] (cf. also Remark 3.5, p. 43). The worst-case query complexity of Observation Pack is $\mathcal{O}(kn^2 + n\log m)$, which is already very close to the known lower bound of $\mathcal{O}(kn^2)$ (and even coincides with the latter assuming $m = 2^{\mathcal{O}(kn)}$). Also, its performance in practice has been observed to be very good, which is witnessed by the fact that it was the algorithm used by the winning entry [94] in the 2010 ZULU competition [58]. In this competition, participants were ranked according to the quality of their inferred hypotheses after a limited number of membership queries and without any equivalence queries (i.e., requiring the participants to approximate these using membership queries).

However, the Observation Pack algorithm suffers from the fact that the length of the queries it generates grows with the length of the counterexamples provided by the teacher.[1] This does not pose a problem in settings where a *cooperative teacher* [175] provides minimal counterexamples. However, such an assumption is rather unrealistic, as equivalence queries often have to be approximated using membership queries (as described in Section 4.2.4). In such settings, techniques guaranteeing minimal counterexamples are usually avoided: they typically require exploring the search space in a breadth-first fashion, resulting in a number of membership queries that is exponential in the exploration depth d (i.e., in $\Omega(k^d)$).

Other sources of counterexamples may exhibit even more extreme properties. Bertolino *et al.* [33] propose a *life-long learning* approach (sketched in Figure 5.1), where inferred models of networked systems are continuously validated by monitoring their live executions. If a divergence between the observed and the predicted behavior is detected, the corresponding execution trace is provided to the learner as a counterexample. Isberner *et al.* [110] however point out that this causes major performance degradations in the learning process, as these counterexamples may consist of tens of thousands of symbols, which is inacceptable due to the fact that the time for realizing a membership query typically grows linearly with its length (cf. also Section 3.2.1).

On a much smaller scale, the example run of Observation Pack in Section 4.2.3 (cf. also Figure 4.10) already hinted at the problem at hand: the provided first counterexample resulted in the discriminator $aaaaaab$ being added to the discrimination tree (cf. Figure 4.10d). In the considered case, the third hypothesis from Figure 4.10e (and thus the discrimination tree from Figure 4.10f) was already the final one, but assuming it were not, every subsequently added transition would possibly have to be tested against the discriminator $aaaaaab$ when sifting it into the tree. Thus, a single long counterexample at an early stage results in long queries throughout the *entire* rest of the learning process.

[1] Notably, the ZULU competition only limited the number of membership queries, not the total number of symbols occurring in them.

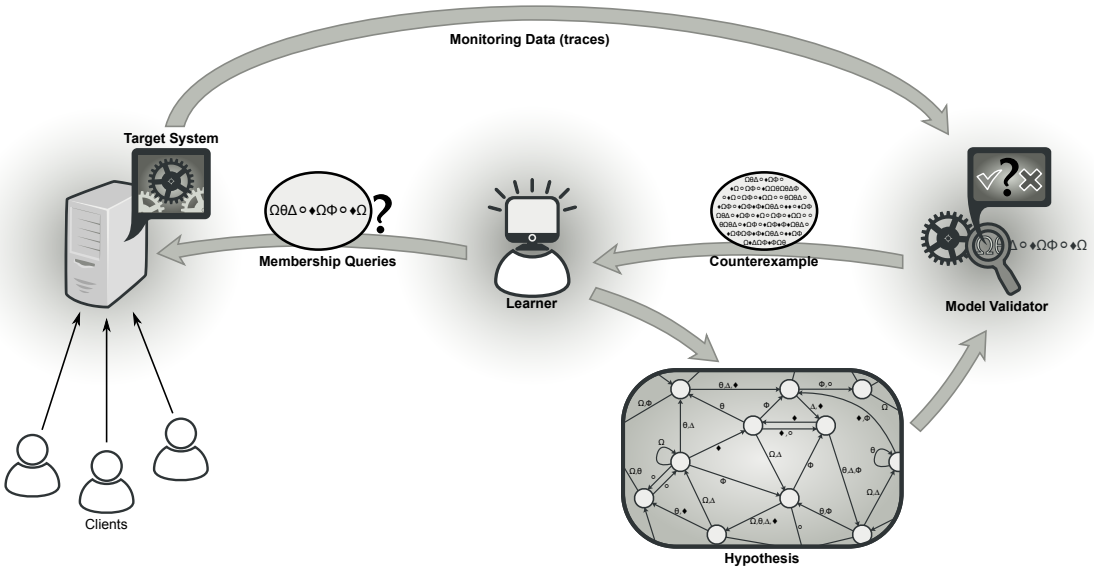


Figure 5.1.: *Life-long learning* approach, proposed by Bertolino *et al.* [33] (source: [110])

In this chapter, we will develop an algorithm, called TTT [110], that overcomes the above deficiencies of the Observation Pack algorithm. It accomplishes this by eagerly attempting to *clean up* its internal data structures, by replacing discriminators extracted from counterexamples (which simply "do the job" of splitting a class) with discriminators which are derived from the transition structure of the hypothesis (and typically are of shorter length). Thus, while long counterexamples might incur some inherent overhead during their analysis itself, the effect on the internal data structures after fully processing a counterexample is the same as if a minimal counterexample had been processed.

The next section starts by describing the idea and design goals on a high level, while the subsequent sections detail on the technical realization. Section 5.3.2 elaborates on an interesting theoretical property of TTT, namely the fact that it is *space-optimal*. The practical evaluation reported on in Section 5.5 furthermore disproves the assumption that the above-sketched process of "cleaning up" incurs some overhead: the evaluation results show that there is no noticeable such overhead even in the presence of minimal counterexamples, and a significant performance gain in the case of non-minimal counterexamples. Notably, this gain can not only be observed when considering the overall number of symbols (reducing the number of which by shortening discriminators was the initial goal of TTT), but a reduction of the number of membership queries can sometimes be observed, too, suggesting that TTT is uniformly superior to other considered algorithms.

5.1. Design Goals and High-level Overview

One of the goals stated in the introduction to this thesis was to formally characterize phenomena in active automata learning, to identify desirable properties, and to use these findings as a basis for developing an efficient algorithm that adequately addresses these phenomena to ensure the

desirable properties.

Clearly, two fundamental properties of black-box abstractions are *closedness* and *determinism* (cf. Definition 3.9, p. 29), as they are preconditions for being able to construct a DFA hypothesis. Interestingly, while the alternated check (and, if they are found to be violated, restoration attempts) of these properties dominates the flow of control of the original L* algorithm due to Angluin [19], nothing comparable can be found in the Observation Pack algorithm (cf. Algorithms 4.6 and 4.7). This is due to superimposed constraints which *guarantee* determinism and closedness: maintaining $\mathcal{U}$ (implicitly given by the spanning-tree hypothesis) as a set of pairwisely inequivalent short prefixes renders the determinism requirement trivial, while the discrimination tree data structure prevents unclosednesses.[2] Note that both are manifestations of minimality requirements: the former means that no superfluous short prefix is ever added to $\mathcal{U}$, while the latter is due to the fact that when sifting a transition into the discrimination tree, only queries that are *necessary* to discriminate between existing classes are posed, thus precluding the possibility of "accidentally" identifying new classes (at least for a binary output domain).

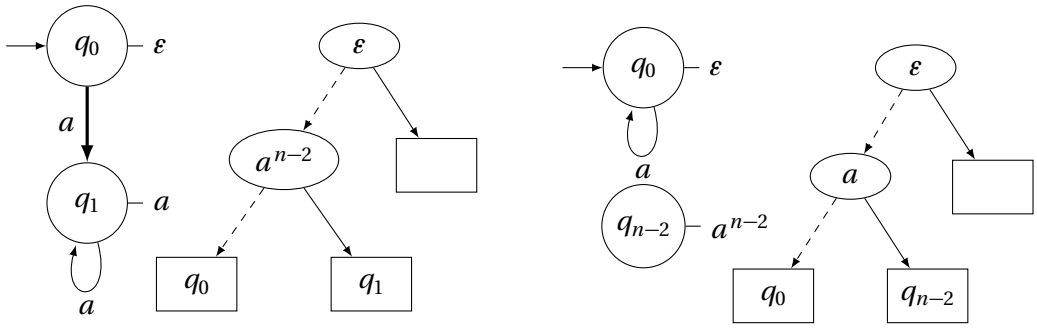
Besides these two *necessary* preconditions, we also identified two *desirable* properties—*reachability consistency* (cf. Definition 3.11, p. 30) and *output consistency* (cf. Definition 3.12, p. 31)—which are not necessary for correctness or being able to construct a hypothesis (Theorem 3.2 furthermore guarantees that they will be satisfied eventually), but rather correspond to a certain *quality* of the hypothesis with respect to the observations. Furthermore, we have identified how these properties can be ensured by enforcing certain syntactical constraints: maintaining $\mathcal{U}$ as a prefix-closed set guarantees reachability consistency, while maintaining semantically suffix-closed characterizing sets ensures output consistency.

Unlike in the above case of closedness and determinism, which are taken care off by the mere choice of data structures, there seems to be a trade-off between the syntactical properties ensuring reachability and output consistency: the algorithm by Kearns and Vazirani [115] ensures semantic suffix-closedness, but short prefixes are no longer maintained in a prefix-closed fashion. In contrast, the Observation Pack algorithm ensures prefix-closedness of $\mathcal{U}$ (by means of the spanning-tree hypothesis), but forgoes (semantic) suffix-closedness of the discriminator sets. For both algorithms, the cause for the violation of the respective properties is their strategy of handling counterexamples (cf. Theorem 3.3 as well as Remarks 3.4 and 3.6, respectively).

5.1.1. Property Restoration

Lemma 3.6 (iii) states how the result of a prefix-based counterexample analysis (or reachability inconsistency analysis) can be exploited to refine a black-box abstraction: adding the prefix $\widehat{u}$ to $\mathcal{U}$ causes non-determinism, which is then eliminated by splitting an equivalence class. Lemma 3.8 (iii) handles the symmetrical case, concerning the result of an output inconsistency analysis: splitting a class using $\widehat{u}$ as discriminator causes an unclosedness, which is eliminated by adding a new prefix to $\mathcal{U}$. From this perspective, it can be argued that the above descrip-

[2] There are two exceptions to this: unclosednesses can occur when learning Mealy machines (cf. Section 3.5.3), as only two identified children are necessary to cause the introduction of an inner node in the first place, but many more children can be discovered *en passant*. The other exception occurs when learning DFAs: the first state with an acceptance value different from that of the hypothesis is also discovered *en passant* (this case is handled in line 22 of Algorithm 4.6), which basically means it is added to the hypothesis by eliminating an unclosedness. However, since this occurs only once during the learning process, and other unclosednesses can in fact not occur when learning DFAs, the above statement is justified when contrasting the situation to observation tables.



(a) Violation of semantic suffix-closedness if prefix-closedness of $\mathcal{U}$ is maintained

(b) Violation of prefix-closedness of $\mathcal{U}$ if semantic suffix-closedness is maintained

Figure 5.2.: Illustration of necessary violations when learning the DFA from Figure 4.7

tion about the two algorithms (Kearns and Vazirani's algorithm and Observation Pack) maintaining closedness and determinism the entire time is somewhat imprecise: unclosedness and non-determinism are momentarily introduced during counterexample analysis, and then eliminated immediately.

Since violations of prefix-closedness of $\mathcal{U}$ or (semantic) suffix-closedness also arise from handling counterexamples, one may ask whether it might be possible to immediately restore those properties as well. Unfortunately, this is generally not the case, at least not *atomically*. Consider the DFA $\mathcal{A}'_n$ depicted in Figure 4.7: if prefix-closedness is to be maintained, the first new short prefix to be added is a. However, distinguishing a from ε requires the discriminator a^{n-2}, which violates semantic suffix-closedness (sketched in Figure 5.2a). On the other hand, maintaining suffix-closedness means that the first discriminator used to split a leaf must be a, which however can only distinguish a^{n-2} from any other short prefix corresponding to a rejecting state, violating prefix-closedness of $\mathcal{U}$ (cf. Figure 5.2b).

Nevertheless, it is possible—and this is the way the TTT algorithm works—to restore the property of (semantic) suffix-closedness[3] after a *sequence* of insertions of new states. The DFA from Figure 4.7 discussed above constitutes the worst-case, as *every* state needs to be added to restore suffix-closedness; however, often a much smaller number of additional states is sufficient to reach a point which allows restoration of suffix-closedness.

5.1.2. Interplay of Data Structures

The fact that the TTT algorithm maintains a suffix-closed set of discriminators allows for storing the overall set of discriminators in a trie (cf. Proposition 4.1). The resulting interplay of data structures is visualized in Figure 5.3: the *spanning-tree hypothesis* (left) maintains information about the access sequences of states, and separates definite (tree) transitions from tentative (non-tree) ones. States and (non-tree) transitions of the hypothesis are associated with (or point

[3] Actually, TTT only ensures that the *overall* set of discriminators is suffix-closed, without the black-box abstraction necessarily being *semantically* suffix-closed. As, due to its other characteristics, the TTT algorithm natively includes a check for output inconsistencies (and elimination of these), output consistency is eventually restored, and neglecting to ensure semantic suffix-closedness makes for a much simpler implementation. We will elaborate on the necessary adaptations for actually restoring and ensuring semantic suffix-closedness in Section 5.2.5.

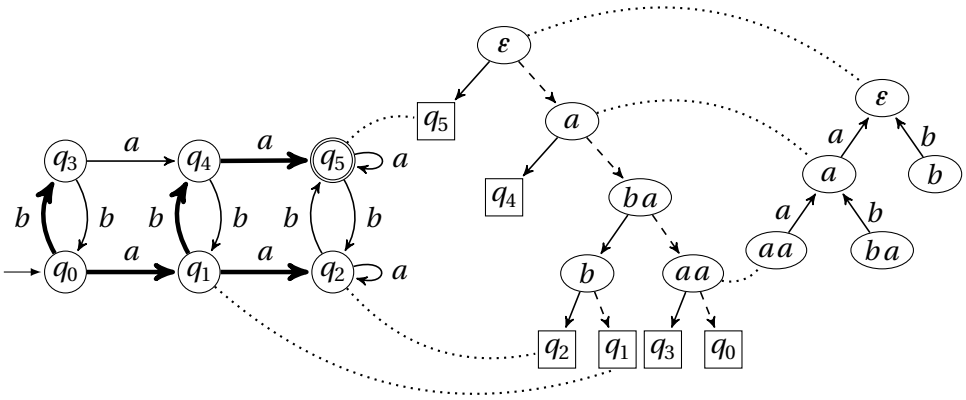


Figure 5.3.: Interplay of data structures in the TTT algorithm (left to right): spanning-tree hypothesis, discrimination tree, suffix trie (source: [100])

to) nodes in the *discrimination tree* (middle). The inner nodes of the discrimination tree in turn correspond to nodes in the *suffix trie* (right), allowing for a compact representation and storage. The combination of these three tree-based data structures—spanning-**T**ree, discrimination **T**ree, and suffix **T**rie—is what gives rise to the name TTT.

5.2. Technical Realization

The TTT algorithm builds on top of the Observation Pack algorithm. In particular, it eliminates excessively long discriminators (as discussed in the previous section) by continuously cleaning up the internal data structures and reorganizing the discrimination tree. As the Observation Pack algorithm has already been discussed in great detail in Section 4.2.3, the description of TTT in this section will thus be kept incremental, i.e., focusing on the additional steps only, some of which however are quite involved.

5.2.1. Temporary and Final Discriminators

When presented with a counterexample, the Observation Pack algorithm analyzes this counterexample to obtain a decomposition $\langle \widehat{u}, \widehat{a}, \widehat{v} \rangle$. It then splits a leaf in the discrimination tree, and labels the new inner node with $\widehat{v}$. This node remains unchanged in the discrimination tree throughout the entire course of the learning process, potentially leading to performance problems due to long queries, as discussed in the introduction of this chapter.

The first steps that TTT takes when being presented with a counterexample are the same as those of Observation Pack (cf. Algorithm 4.7): it *splits* a node in the discrimination tree, using the obtained suffix $\widehat{v}$ as the new discriminator. However, since $\widehat{v}$ usually violates suffix-closedness of the discriminator set, meaning it cannot be added to the suffix trie by adding a single node, it is marked as *temporary*. In contrast, nodes labeled with a discriminator that is already integrated into the suffix trie are called *final*. In terms of data structures, we assume that every (inner) node $n \in \mathcal{N}_{\mathcal{T}}$ has a Boolean flag $n.temp$ indicating whether n is temporary. Whenever a leaf is split, its flag is set to **true**. Furthermore, we assume that we always have $r_{\mathcal{T}}.temp = \textbf{false}$, i.e., the root

Algorithm 5.1 "Soft" sifting in a discrimination tree

Require: Discrimination tree $\mathcal{T}$, node $n \in \mathcal{N}_\mathcal{T}$, prefix $u \in \Sigma^*$, output function λ (implicit)

```
1: function soft-sift_T(n, u)
2:     while n ∈ I_T and ¬n.temp do
3:         o ← λ(u, n.discriminator)
4:         n ← n.children[o]
5:     end while
6:     return n
7: end function
```

discriminator ε is guaranteed to be final.

The term "temporary" (which we will apply to both a discriminator and its corresponding node) here refers to the fact that the respective discriminator will subsequently be replaced ("finalized") with another discriminator. In fact, we will see that even the entire topology of subtrees rooted at temporary inner nodes may change.

Soft Sifting

To avoid posing membership queries involving temporary (and thus potentially long) discriminators as suffixes, the TTT algorithm modifies the behavior of the CLOSETRANSITIONS procedure (cf. Algorithm 4.6) such that the sifting of transitions is only continued until the first temporary discriminator is encountered. This is also referred to as *soft sifting*. Thus, as a result, the hypothesis might still be non-deterministic (in particular, the incoming non-tree transitions of the state q_{old} being split remain unmodified, and point to the newly introduced temporary inner node).

The logic of soft-sifting is given as Algorithm 5.1. The modified procedure CLOSETRANSITIONS-SOFT can thus be obtained from the Observation Pack version, with the only modification that *soft-sift*$_\mathcal{T}$ is called in line 20 of Algorithm 4.6. Note that, in contrast to the regular *sift*$_\mathcal{T}$ function, the result of soft sifting is no longer guaranteed to be a leaf, which motivates the additional check in the **if** statement in line 22 of Algorithm 4.6, handling newly discovered states.

5.2.2. Discriminator Finalization – Simple Case

After calling CLOSETRANSITIONS-SOFT, the hypothesis usually remains non-deterministic, but every non-tree transition points to either a leaf, or a temporary inner node with a final (non-temporary) parent.

Let us introduce the context of a *block subtree*, or simply *block*. A block subtree is a maximal subtree containing neither final inner nodes nor unlabeled (meaning: without an associated state) leaves, i.e., it contains only labeled leaves and temporary inner nodes. We also identify the block subtree with all the states corresponding to the leaves it contains (in which case we usually use the term "block", though we make no strict distinction). Since every labeled leaf is either part of a (bigger) block, or constitutes a singleton block, it is obvious that the set of all blocks forms a partition of $Q_\mathcal{H}$.

Consider the situation depicted in Figure 5.4. The target DFA is the same that was used for the example run of Observation Pack (cf. Figure 4.10), and the figure depicts the data structures

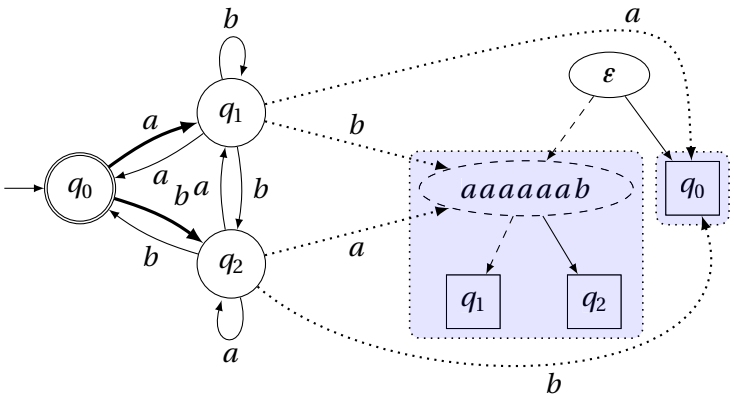


Figure 5.4.: TTT data structures after introduction of temporary discriminator and soft closing. The dashed outline marks the inner node as temporary, while dotted lines represent the *tgt_node* pointers. Rounded rectangles indicate blocks

while the counterexample $w = baaaaaab$ is being processed. As stated above, the first steps are the same as in Observation Pack: a new state with access sequence b is introduced, and the suffix $aaaaaab$ is used to split the leaf formerly corresponding to q_1. However, in TTT this discriminator is marked as *temporary*, which is visualized by the respective inner node having a dashed outline.

Blocks are visualized as rounded rectangles enclosing the respective subtree. In Figure 5.4, there are two blocks: a singleton block, containing only the leaf corresponding to q_0, and a non-trivial block, which contains q_1, q_2 and the temporary discriminator. Blocks also determine the "granularity" of non-determinism in the hypothesis: we have $\delta_{\mathcal{H}}(q_1,b) = \delta_{\mathcal{H}}(q_2,a) = \{q_1,q_2\}$, whereas the a-transition of q_1 and the b-transition of q_2 point to a singleton block and are thus deterministic. The outgoing transitions of q_0 are both tree transitions and therefore always deterministic, even though they point to states in a non-singleton block.

For now, let us simply assume that *every* temporary inner node is part of a block, meaning that every proper ancestor of a block root is final. This is obviously the case in Figure 5.4, and we will later discuss how this can be ensured in general. This means that the lowest common ancestor from any two nodes in *distinct* block subtrees is necessary final. Recalling the white-box discrimination tree computation presented in Section 4.1.4, the path to replacing the temporary discriminator with a final one becomes pretty obvious.

The dotted lines in Figure 5.4 represent the *tgt_node* pointers of the outgoing transitions of q_1 and q_2. As we can easily see, for both a and b, the corresponding transitions of both states point into different blocks. Since the respective target nodes are separated by the final discriminator ε, $\varepsilon \cdot a$ and $\varepsilon \cdot b$ can both be used to distinguish q_1 and q_2. Assume that we choose $\varepsilon \cdot a = a$ as the final discriminator. The fact that $\lambda(\lfloor q_1 \rfloor, a) = \lambda(a,a) = 1$ and $\lambda(\lfloor q_2 \rfloor, a) = \lambda(b,a) = 0$ can be derived from the target nodes of the a-transitions of q_1 and q_2. Therefore, no additional membership queries are required to construct the discrimination tree shown in Figure 5.5. Note that the role of a for separating q_1 and q_2 is not exactly the same as that of the temporary discriminator $aaaaaab$: in Figure 5.4, q_1 was the 0-child of its parent and q_2 the 1-child, whereas it now is the other way round. The corresponding hypothesis (shown in the left of Figure 5.5, after closing all open transitions) is now deterministic, as there are no more temporary discriminators.

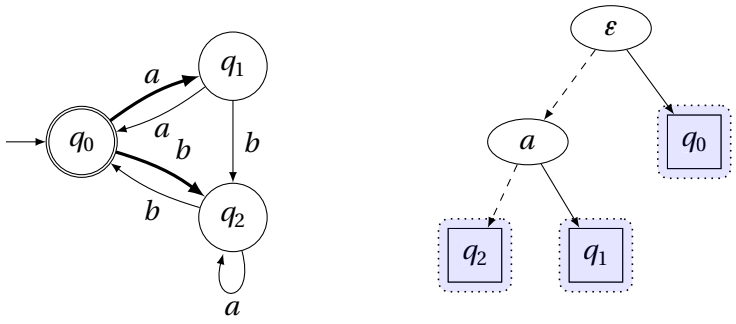


Figure 5.5.: Closed hypothesis and discrimination tree after replacing the temporary discriminator $aaaaaab$ in Figure 5.4 with the final discriminator a

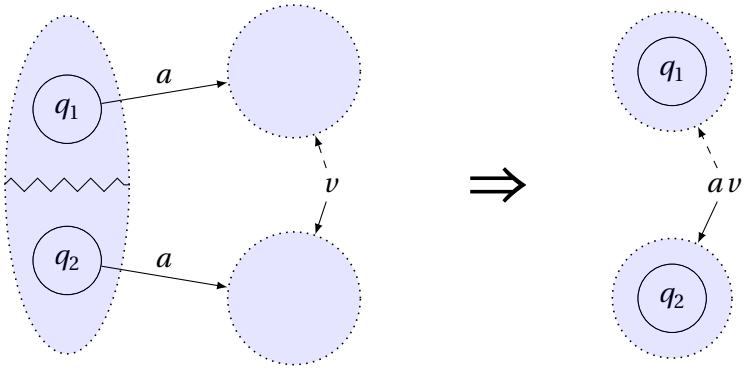


Figure 5.6.: Abstract visualization of discriminator finalization

The abstract idea behind discriminator finalization is illustrated in Figure 5.6: the a-transitions of q_1 and q_2 in the same block (enclosing ellipse) point into different blocks, which in turn are separated by the final discriminator v. This allows to split the block containing q_1 and q_2 into two blocks which are separated by $a\,v$.

5.2.3. Output Inconsistencies and Subsequent Splits

Unfortunately, discriminator finalization is not always as easy as the previous subsection might suggest. There may be situations in which it is impossible to finalize any discriminator, since all outgoing a-transitions (for any $a \in \Sigma$) of nodes in one block point into the same block. This can be formalized as follows.

Define by $\pi(\mathcal{T})$ the set of all blocks in a discrimination tree $\mathcal{T}$, where a block is defined as in the previous Section 5.2.2. Clearly, $\pi(\mathcal{T})$ forms a partition of $Q_{\mathcal{H}}$. The above condition that no final discriminator can be determined can be characterized formally via

$$\forall B \in \pi(\mathcal{T}) : \forall a \in \Sigma : \exists B' \in \pi(\mathcal{T}) : \delta_{\mathcal{H}}(B,a) \subseteq B', \tag{5.1}$$

where $\delta_{\mathcal{H}}$ denotes the (non-deterministic!) transition function of $\mathcal{H}$ lifted to sets of states, as introduced in Remark 2.1.

An important observation is that (5.1) still holds if the extended transition function and words $w \in \Sigma^*$ instead of single symbols are considered. Furthermore, the fact that the root of $\mathcal{T}$ is

always final ensures that $\sim_{\pi(\mathcal{T})}$ saturates $F_{\mathcal{H}}$ (i.e., $\forall B \in \pi(\mathcal{T}) : B \subseteq F_{\mathcal{H}} \vee B \cap F_{\mathcal{H}} = \emptyset$). As a consequence, the non-determinism in $\mathcal{H}$ does not cause any uncertainty wrt. whether a word $w \in \Sigma^*$ is accepted or not, i.e.,

$$\forall B \in \pi(\mathcal{T}) : \forall w \in \Sigma^* : \delta_{\mathcal{H}}(B, w) \subseteq F_{\mathcal{H}} \vee \delta_{\mathcal{H}}(B, w) \cap F_{\mathcal{H}} = \emptyset.$$

Thus, it makes sense to define state output functions $\lambda^q_{\mathcal{H}}$ for $q \in Q_{\mathcal{H}}$ regardless of the non-determinism, and all states within the same block are equivalent in the sense that their state output functions are equal. The intuition behind this is that due to (5.1), any two "determinizations" of $\mathcal{H}$ obtained by arbitrarily choosing one of the possible targets for each transition would be equivalent. This implies that even if all transitions were fully closed using "hard" sifting (i.e., not stopping at temporary nodes), the output functions would not change, resulting in a *non-canonical* deterministic hypothesis computing the same output function as the current non-deterministic one.

Addressing Output Inconsistencies

The fact that we can assign output functions to states regardless of non-determinism enables us to reason about output inconsistencies in our hypothesis. Since every pair of distinct states $q \neq q'$ in some block $B \in \pi(\mathcal{T})$ is separated by their (temporary) lowest common ancestor, but their state output functions agree on all possible arguments, this necessarily means that one of them must constitute an output inconsistency. Formally, this means that whenever (5.1) holds, then also

$$\forall B \in \pi(\mathcal{T}) : |B| > 1 \Rightarrow \exists q \in B : \exists (v, o) \in Sig_{\mathcal{T}}(q) : \lambda^q_{\mathcal{H}}(v) \neq o.$$

Such an output inconsistency can be addressed using the techniques from Section 3.3.4. Analyzing an output inconsistency in the simplified way described in Remark 3.5 (p. 43) however requires a deterministic transition function. Thus, whenever an output inconsistency (q, v) needs to be analyzed, the visited transitions need to be determinized on-the-fly by "hard" sifting.[4]

As a result, new states with new transitions (that are then softly closed) are added to the hypothesis, along with new temporary discriminators in the discrimination tree. Note that every temporary discriminator is inserted by splitting a leaf (which is by definition part of a block), thus resulting in the block being *augmented*, and preserving the above-stated property that no temporary discriminators occur outside of block subtrees. Since every newly introduced state is guaranteed to be distinct from every existing state, Theorem 3.2 (p. 33) guarantees that eventually a "correct" (if all transitions were closed using "hard" sifting) hypothesis is obtained, that is furthermore canonical. Since (5.1) implies that *no* determinization (in the above sense) of the hypothesis is canonical, it follows that a finite number of subsequent splits must eventually cause (5.1) to become violated.

Let us give an example to illustrate the process. Assume that the target DFA is the one shown in the left of Figure 5.3. The initial hypothesis and discrimination tree are shown in Figures 5.7a and 5.7b, respectively. After being provided the counterexample $w = abbab$, the first refinement step results in the non-deterministic hypothesis shown in Figure 5.7c, with the corresponding discrimination tree from Figure 5.7d. Since there is only a single block in the discrimination tree, it is obvious that no finalization step is possible.

[4] When using a non-local search heuristic such as binary or exponential search, it makes sense to do this lazily. That is, for determining the state reached by v from q in the *determinized* hypothesis, one can check whether there is an index $i, 1 \leq i < |v|$, such that $|\delta_{\mathcal{H}}(q, v_{1..i})| = 1$, and start from the largest such index.

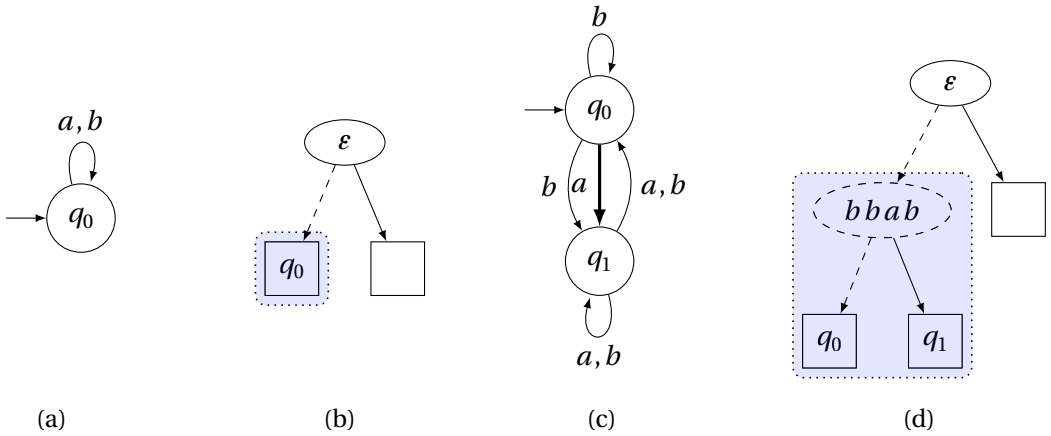


Figure 5.7.: Hypotheses and discrimination trees during a run of TTT on the DFA shown in Figure 5.3

Analyzing the output inconsistency constituted by $(q_1, b\,b\,a\,b)$ results in the leaf corresponding to q_0 being split, using $b\,a\,b$ as the temporary discriminator, and the introduction of a new state, q_4, with access sequence $a\,b$.[5] As a result, the accepting state q_5 is discovered *en passant*, and assigned the access sequence $a\,b\,a$. The corresponding hypothesis and discrimination tree are shown in Figure 5.8, where all missing transitions non-deterministically point to $\{q_0, q_1, q_4\}$.

5.2.4. Discriminator Finalization – Complex Case

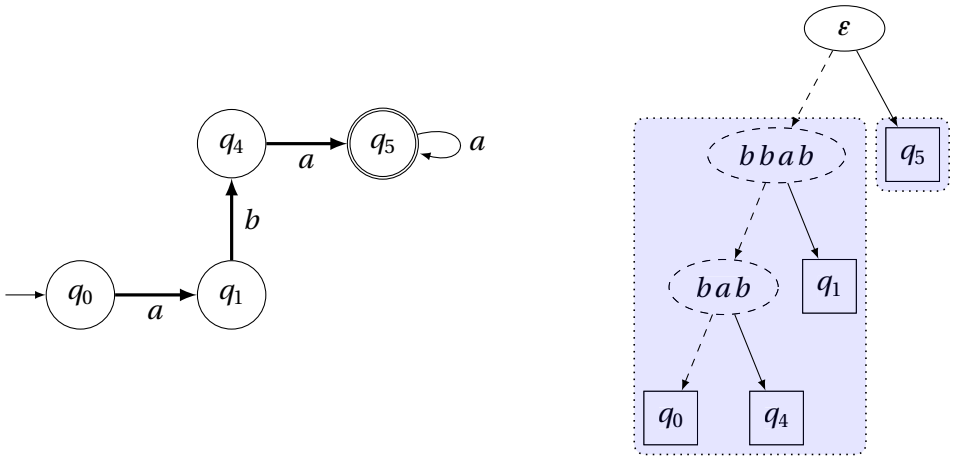
If, after a number of subsequent splits, condition (5.1) is violated, this again allows us to finalize discriminators. However, we are now faced with a more complex situation than the one described in Section 5.2.2: blocks may now contain more than just two states, which means in particular that block subtrees may contain more than one temporary inner node. Even worse, there may be non-tree transitions that point to a proper descendant of a block root, as hard sifting might have become necessary during counterexample analysis.[6]

First, let us reconsider what the violation of (5.1) means: there exists a block B and a symbol $a \in \Sigma$, such that for $q, q' \in B$ with $q \neq q'$, the a-transitions of q and q' point into different blocks. Consequently, the lowest common ancestor of the corresponding transitions' *tgt_node*'s is a *final* inner node. This furthermore implies that the lowest common ancestor of the *tgt_node*'s of *all* a-transitions of states in B is a final inner node, since every ancestor of a final inner node is also final.

The situation we are now facing is thus very similar to the one during discrimination tree computation in the white-box setting, where the discrimination tree is augmented using the $\text{SPLIT}_{\text{SINGLE}}$ function (cf. Algorithm 4.4 and Algorithm 4.5): if v' is the (final!) discriminator of the common LCA of all a-successors of states in a block B, then $v = a\,v'$ is a discriminator that preserves suffix-closedness of the discriminator set, and can be used to split B into two non-

[5] For simplicity, we use the corresponding state names from the final hypothesis shown in Figure 5.3, instead of assigning contiguous indices.

[6] Note that as a result, non-tree transitions may point to arbitrary nodes within a block subtree: closing them using hard sifting results in them pointing to a leaf, but this leaf may subsequently be split.



(a) Hypothesis after subsequent split (b) Discrimination tree after subsequent split

Figure 5.8.: TTT data structures after addressing the output inconsistency $(q_1, b\,b\,a\,b)$ from Figure 5.7 by subsequently splitting q_0. Non-deterministic transitions point to $\{q_0, q_1, q_4\}$ and are omitted for the sake of clarity

empty blocks.

The general idea of discriminator finalization is to "replace" the discriminator of the root of a block subtree using a final discriminator obtained in the above fashion. The root is chosen to maintain the property that every descendant of a temporary node is also temporary or a leaf. Accomplishing this is however not trivial: the final discriminator v usually partitions the states in the block in a way that is different from that of the temporary block root discriminator $\widehat{v}$. In fact, there might be situations where $\widehat{v}$ remains necessary to separate some states in one (or even both) of the sub-blocks resulting from the split using v.

Discriminator Replacement

The general strategy for replacing discriminators at the root of block subtrees bears some resemblance to the SPLIT$_{tree}$ approach (cf. Figure 4.5), which is based on "carving out" subtrees, and can be described as follows:

1. For each state q labeling a leaf in the block subtree, perform a membership query $\lambda(\lfloor q \rfloor, v)$ to determine whether it needs to be in the 0- or the 1-subtree of the new final inner node.[7]

2. Carve out a subtree containing only the leaves for which the membership query returned 0. This can be accomplished by marking all such leaves, and propagate the marking all the way up to the block root. Then, discard all unmarked nodes, and replace all inner nodes with a single child with this child.

3. The resulting subtree forms the 0-subtree of the new final inner node.

[7] Since $v = a\,v'$, where v' is the suffix labeling the least common ancestors of the target nodes of all a-transitions, a membership query is not really required here. Instead, it suffices to determine the label of the subtree of this LCA into which the a-transition of q points.

Algorithm 5.2 TTT-REPLACE-BLOCKROOT: Discriminator finalization in the TTT algorithm

```
1:  procedure TTT-REPLACE-BLOCKROOT(𝒯, r_B, v)
2:      Initialize mark as a map from nodes to sets of Booleans
3:      Initialize inc_0, inc_1 as maps from nodes to sets of transitions
4:      Initialize state_0, state_1 as maps from leaves to states
5:      mark[r_B] ← {0,1}                ▷ ensure marks are not propagated beyond block root
6:      for n ∈ Desc(r_B) do                              ▷ iterate over all nodes in the block
7:          for t ∈ n.incoming do          ▷ compute resulting subtree for incoming transitions
8:              o ← λ(t.aseq, v)
9:              inc_o[n] ← inc_o[n] ∪ {t}   ▷ record inc. transitions of the o-subtree version of n
10:             MARK(n', o)      ▷ mark node with o if at least one transition is in the o-subtree
11:         end for
12:         if n ∈ 𝓛_𝒯 then                                                  ▷ n is a leaf
13:             q ← l.state
14:             o ← λ(⌊q⌋, v)
15:             state_o[n] ← q                ▷ record state for the o-subtree version of n
16:             MARK(n, o)                    ▷ mark node with o if its state is in the o-subtree
17:         end if
18:     end for
19:     𝒯_0 ← EXTRACT(r_B, 0)                                    ▷ extract the subtrees
20:     𝒯_1 ← EXTRACT(r_B, 1)
21:     𝒯' ← MAKE-INNER(v, 𝒯_0, 𝒯_1)      ▷ ... make them children of a node with discriminator v
22:     REPLACE-NODE(𝒯, r_B, 𝒯')                  ▷ ... and replace the entire block subtree
23: end procedure
```

4. Repeat 2. and 3. for 1 instead of 0.

5. Replace the block root with an inner node labeled with the final discriminator v, and the 0- and 1-subtrees as described above as children.

While the approach outlined above is already quite complex, it is insufficient because it does not address the non-tree transitions pointing to *proper* descendants of the block root, which might have been introduced by hard sifting during counterexample analysis. Of course, it would be possible to *reset* all these transitions by letting them point to the new final inner node that replaces the block root (and softly close them by sifting them down one level), but that would also mean to throw away the information gained through hard sifting, which is unacceptable if redundancy freeness is our aim.

This problem calls for an even more involved approach, for which we show the pseudocode in Algorithms 5.2 through 5.4, and that we now want to discuss. The procedure TTT-REPLACE-BLOCKROOT, given as Algorithm 5.2, constitutes the entry point to discriminator replacement in the TTT algorithm. It maintains the following maps to realize the subtree extraction: *mark* maps nodes in the block subtree to subsets of $\mathbb{B}$, thereby representing the marks for the extraction of the 0- and the 1-subtree.[8] Like in SPLIT_{tree} (cf. Algorithm 4.5), it will be main-

[8] An alternative way, that more closely resembles the way *mark* is used in Algorithm 4.5, would be to maintain two separate maps $mark_0$ and $mark_1$, mapping nodes to Booleans.

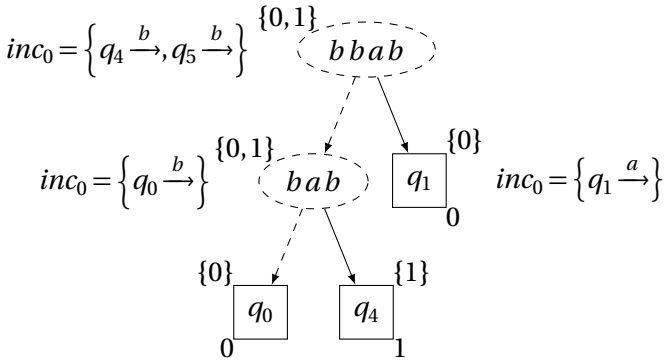


Figure 5.9.: Block subtree from Figure 5.8 after computing inc_o, $state_o$ and *mark* values

tained in such a way that marks are always propagated upwards in the block subtree, i.e., for $o \in \mathbb{B}$ and a node n in the subtree that is not the block root r_B, $o \in mark[n]$ (we also say that "n is o-marked") implies $o \in mark[n.parent]$. Marking the block root (line 5) ensures that marks are not propagated beyond this node.

Furthermore, maps inc_o and $state_o$, $o \in \mathbb{B}$, are maintained, mapping nodes to sets of non-tree transitions and leaves to states, respectively. Their significance can be explained as follows. A node n in the discrimination tree, in case it is a leaf, stores a reference to the state it corresponds to ($n.state$), and we furthermore assume that any node stores a set of all its incoming non-tree transitions (referred to via $n.incoming$). The extraction process can be thought of as creating two copies of the block subtree, one for each possible output value $o \in \mathbb{B}$, and the semantic information stored with a node (i.e., the incoming transitions and, in the case of leaves, the state) are then reassigned to *one* of the copies of this node—-which one depends on the output behavior wrt. v. Since the extracted subtrees are created on-the-fly, the purpose of these four maps is to store the data for these *copies* of nodes yet to be created.

The maps are filled with data in the **for** loop iterating over all nodes in the block subtree (lines 6–18). Lines 7–11 take care of preliminarily assigning incoming transitions to the future copy by adding them to the $inc_o[n]$ set, and lines 12–16 take care of assigning the states labeling leaf. Whenever any of these are determined to correspond to the future o-copy of n ($o \in \mathbb{B}$), n is marked with o. Thus, whenever a node n in the block subtree is o-marked, this indicates that itself or one of its descendants has a non-empty inc_o set or a non-**nil** $state_o$ value. The **for** loop in lines 6–18 can thus be thought of as a preprocessing step for the subsequent subtree extraction.

Example. Continuing our example from Figure 5.8, it is obvious that the a-transition of q_4 points to a different block (namely the singleton $\{q_5\}$) than the a-transitions of the other two states in the same block. Thus, $a \cdot \varepsilon = a$ is a final discriminator that can split the bigger block.

In the following, we assume for the purpose of demonstration that the b-transition of q_0 points to the temporary inner node labeled with $b\,a\,b$ and the a-transition of q_1 points to the leaf associated with q_4. All other non-tree transitions point to the block root. The result of computing the inc_o and $state_o$ values (and thus the markings) for the final suffix $v = a$ is shown in Figure 5.9. The non-empty inc_o sets of each node are shown next to the node; we write $q \xrightarrow{a}$ to refer to the a-transition of q. One of the bottom corners of each leaf is annotated with the value corresponding to its state, that is, leaf l is annotated with $o \in \mathbb{B}$ if and only if $state_o[l] \neq$ **nil**. Finally, one of the top corners of each node is annotated with the corresponding *mark* set.

Algorithm 5.3 Helper functions for TTT-REPLACE-BLOCKROOT (Algorithm 5.2)

Require: Node n, output value $o \in \mathbb{B}$, *mark* mapping (implicit)
Ensure: n and all its ancestors in the block subtree have an o mark

```
1: procedure MARK(n, o)
2:     while o ∉ mark[n] do                ▷ propagate mark all the way up to the block root
3:         mark[n] ← mark[n] ∪ {o}
4:         n ← n.parent
5:     end while
6: end procedure
```

Require: Node n, output value $o \in \mathbb{B}$, $state_o$ and inc_o mappings (implicit)
Ensure: Subtree containing only o-marked nodes is returned

```
7: function EXTRACT(n, o)
8:     if n ∈ L_T then                                              ▷ n is a leaf
9:         if state_o[n] ≠ nil then                  ▷ corresponding state is in o-subtree
10:            res ← MAKE-LEAF(state_o[n])              ▷ create the o-subtree version of n
11:        else                          ▷ an incoming non-tree transition is in the o-subtree
12:            return CREATE-NEW(n)                                  ▷ see Algorithm 5.4
13:        end if
14:    else                                                         ▷ n is an inner node
15:        c_0 ← n.children[0], c_1 ← n.children[1]
16:        if o ∈ mark[c_0] ∧ o ∈ mark[c_1] then          ▷ both children are o-marked
17:            T_0 ← EXTRACT(c_0, o)          ▷ therefore, n is necessary in the o-subtree
18:            T_1 ← EXTRACT(c_1, o)
19:            res ← MAKE-INNER(n.discriminator, T_0, T_1)      ▷ create o-subtree version of n
20:        else if o ∈ mark[c_0] then      ▷ only the 0-child is marked (i.e., n is unnecessary)
21:            inc_o[c_0] ← inc_o[c_0] ∪ inc_o[n]                ▷ incoming transitions "fall through"
22:            return EXTRACT(c_0, o)
23:        else if o ∈ mark[c_1] then          ▷ only the 1-child is marked (symmetrical case)
24:            inc_o[c_1] ← inc_o[c_1] ∪ inc_o[n]
25:            return EXTRACT(c_1, o)
26:        else          ▷ both children are unmarked (i.e., n has an o-incoming transition)
27:            return CREATE-NEW(n)                                  ▷ see Algorithm 5.4
28:        end if
29:    end if
30:    res.incoming ← inc_o[n]      ▷ res is the o-subtree version of n, update inc. transitions
31:    return res
32: end function
```

Algorithm 5.4 CREATE-NEW helper function for EXTRACT (cf. Algorithm 5.3)

Require: Node n, output value $o \in \mathbb{B}$
Ensure: Leaf with a newly created state (from one of the o-incoming transitions of n) is returned

```
1: function CREATE-NEW(n, o)
2:     t ← choose(inc_o[n])          ▷ choose any transition (e.g., with shortest access sequence)
3:     q ← MAKETREE(t)               ▷ convert t into a tree transition, resulting in new state q
4:     res ← MAKE-LEAF(q)            ▷ create leaf for q, which is the o-subtree version of n
5:     res.incoming ← inc_o[n] \ {t} ▷ update incoming non-tree transitions
6:     return res
7: end function
```

Subtree Extraction

The recursive EXTRACT function, shown in lines 7–32 of Algorithm 5.3, is a considerably more complex version of the one presented in the context of SPLIT_{tree} (cf. Algorithm 4.5). It creates, for a given output value $o \in \mathbb{B}$, an extracted version of the block subtree on-the-fly. While most of the algorithm is straightforward and almost self-explanatory, we want to emphasize two aspects.

It may be the case that a leaf is o-marked ($o \in \mathbb{B}$), but the corresponding $state_o$ value is **nil** (lines 11–13). This means there is at least one transition in its inc_o set (we call such a transition an o-incoming transition), and the behavior of these transitions wrt. the new suffix v is observably distinct from any state in the hypothesis: v separates them from the state associated with this leaf, and the temporary and final discriminators in the discrimination tree separate them from all other states in the hypothesis. This calls for the introduction of a new state, which is realized by the CREATE-NEW procedure shown as Algorithm 5.4. A new state is created by converting one of its o-incoming non-tree transitions into a tree transition, similar to an *en passant* discovery of a new state while closing transitions (cf. Algorithm 4.6). Another manifestation of this phenomenon is when an inner node is o-marked, but none of its children are. Again, this necessarily implies that its inc_o set is non-empty, and the introduction of a new state is required for the same reason as above (line 27).

The second aspect we want to highlight is that it may be the case that an inner node is o-marked, but only one of its children is o-marked as well. This basically means that a copy of the inner node is not necessary in the extracted subtree (as it would only have a single child), with the consequence that the respective o-incoming transitions simply "fall through" to its marked parent. This is realized by adding them all to the inc_o set of the marked child (lines 21 and 24 of Algorithm 5.3).

Example. In Figure 5.9, we have shown the block subtree from Figure 5.8b annotated with the *mark*, $state_o$ and inc_o values of its nodes. Since there is only a single state and no transition that is marked with 1, it is obvious that the extracted 1-subtree only consists of a single leaf associated with q_4. We therefore want to investigate the extraction process of the 0-subtree in detail.

The left of Figure 5.10 shows the block subtree after removing all nodes whose *mark* set did not contain 0. The sets next to nodes are the inc_0 sets. The inner node labeled with $b\,a\,b$ has only one child, and can thus be eliminated. The result of doing so is shown in the right of Figure 5.10. As a consequence, the b-transition of q_0 which formerly pointed to this inner node is reassigned to the leaf corresponding to q_0.

The extracted 0- and 1-subtrees are then integrated into the overall discrimination tree, by

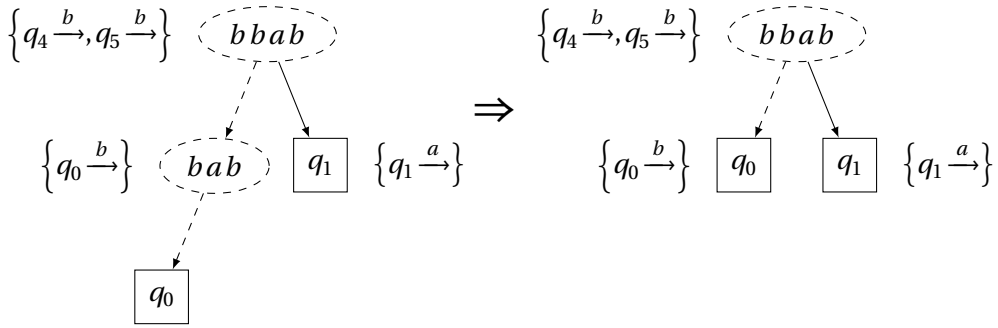


Figure 5.10.: Extraction of the 0-subtree from Figure 5.9

replacing the block root with a final inner node labeled with a and the extracted subtrees as its children. The resulting discrimination tree is shown in Figure 5.11a, the sets next to nodes correspond to the incoming transition sets.

Since q_4 has now been separated from q_0 and q_1, another final discriminator can be obtained: the b-transition of q_1 points to q_4, while the b-transition of q_0 points into the block containing q_0 and q_1. Since both blocks are separated by a, the final discriminator ba can be used to replace the remaining temporary one. Carrying out the replacement as described in this section results in the discrimination tree shown in Figure 5.11b and the (deterministic) hypothesis shown in Figure 5.11c.

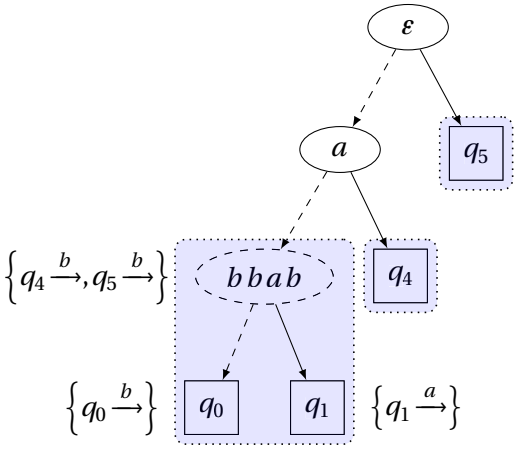
5.2.5. Restoring Semantic Suffix-Closedness

We have remarked in Section 5.1.1 that the version of TTT that we have presented so far actually only maintains the discriminators as elements of a suffix-closed set, but does not necessarily maintain *semantic* suffix-closedness as defined in Definition 3.13 (p. 31). Clearly, this is not due to discriminator finalization: computing the lowest common ancestor of all a-successors of nodes in a subtree, and obtaining the new discriminator by concatenating a and the LCA's discriminator preserves semantic suffix-closedness for all states *currently* in the subtree in the same way as in the white-box scenario (cf. Section 4.1.5). However, adding new states during counterexample analysis (i.e., when splitting leaves, cf. Algorithm 4.7, or from incoming transitions during subtree extraction, cf. Algorithm 5.4) might violate semantic suffix-closedness, as the outgoing transitions of a newly added state have not been tested before.

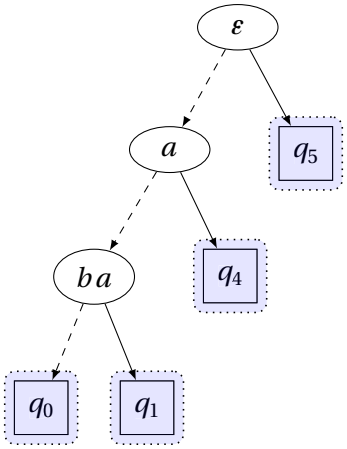
To restore semantic suffix-closedness, it is crucial to first strengthen our property according to our observation from the proof of Lemma 4.2 (p. 73): for every final inner node n labeled with discriminator av, there exists a final inner node n' labeled with v such that *every* outgoing a-transition of a state in the subtree rooted at n points into the subtree rooted at n'.

Let us now consider the case that a state q_{new} is newly added to the hypothesis (and thus a leaf l to the discrimination tree $\mathcal{T}$) during counterexample analysis, which violates the above property. That is, there exists a final inner node n among the ancestors of l that is labeled with av, but the a-transition of q_{new} points to a node n'' that is not a descendant of the node n' (labeled with v) as defined above. Assume that n is chosen such that it is the topmost node in the tree for which the new state violates the property.

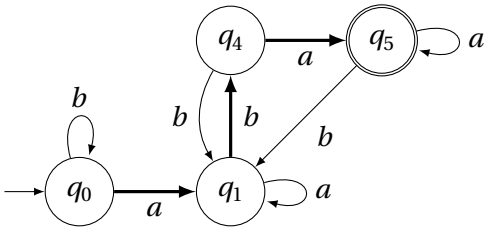
Let n''' be the lowest common ancestor of n' and n'', and assume it is labeled by v'. Then, the



(a) Discrimination tree after first finalization



(b) Discrimination tree after second finalization



(c) Corresponding deterministic hypothesis

Figure 5.11.: Integration of the extracted subtrees into the discrimination tree, subsequent finalization, and corresponding hypothesis a

a-transition of q_{new} points into one of the child subtrees of n''', while the a-transitions of every other state in the subtree rooted at n point into the other subtree. This means that $a\,v'$ can be used to separate q_{new} from all other states in the subtree rooted at n.

Restoring semantic suffix-closedness thus requires inserting a new final inner node labeled with $a\,v'$ *above* n. The required restructuring of the tree can be handled as described in the previous subsections. Note that this means that the final part of the tree no longer grows monotonically. However, since all affected states but q_{new} will be in one of the child subtrees of the newly added node, no final discriminator becomes obsolete (only the temporary one used to separate q_{new} from q_{old} does). Furthermore, the handling of incoming non-tree transitions in Algorithm 5.3 might introduce new states, which again might violate semantic suffix-closedness.

The above outlines a clear approach for maintaining semantic suffix-closedness, but introduces considerable implementation overhead since the final parts of discrimination trees no longer grow monotonically. Besides, we will see in the next subsection that a check for output inconsistencies is performed continuously anyway, which is why maintaining semantic suffix-closedness is not necessary to ensure output consistency. In the following, we will thus only consider versions of TTT that omit the above procedure of explicitly restoring semantic suffix-closedness for the sake of simplicity.

5.3. The Complete Algorithm

Assembling the entire TTT algorithm from the steps discussed in the previous subsections is now relatively easy. The most important observation is that, whenever the property (5.1) defined on p. 94 is violated, a finalization step is possible, and whenever it holds (and there still are temporary discriminators), an output inconsistency must be present. This output inconsistency can then, in the next loop iteration, be analyzed as described in Section 3.3.4, leading to a split as in the case of Observation Pack (cf. Algorithm 4.7).

The actual refinement step is given as Algorithm 5.5, and can be described as a non-strict alternation of output inconsistency analysis and discriminator finalization steps, preferring the latter whenever possible. The initialization phase for TTT is not shown separately, as it is the same as for Observation Pack (cf. Algorithm 4.6).

5.3.1. Complexity

Let us now take a look at the asymptotic complexities of the TTT algorithms. The parameters n, k and m are defined as described in Section 3.2.1.

Query Complexity. The worst-case query complexity is the same as that for the Observation Pack algorithm, i.e., $\mathcal{O}(kn^2 + n\log m)$: each of the kn transitions of the hypothesis eventually needs to be sifted down the entire tree, which has a worst-case depth of $n-1$. Note that this sifting can occur either explicitly, in a call of CLOSETRANSITIONS-SOFT, or implicitly in the preparation for discriminator finalization (i.e., in REPLACE-BLOCKROOT, cf. Algorithm 5.2). In both cases, a single membership query is performed per state or transition for each final discriminator that is added to the ancestor set (i.e., per level in the tree).

Furthermore, no more than $n-1$ counterexample analysis steps are necessary, and due to states having unique representatives, the query complexity for this is $\mathcal{O}(n\log m)$ (cf. Proposition 3.3, p. 44). While the need of "hard" sifting might arise during counterexample analysis,

Algorithm 5.5 TTT-REFINE: Refinement step of the TTT algorithm

Require: Current hypothesis $\mathcal{H}$, corresponding discrimination tree $\mathcal{T}$, counterexample w
Ensure: Refined hypothesis $\mathcal{H}$ and discrimination tree $\mathcal{T}$

procedure TTT-REFINE($\mathcal{H}, \mathcal{T}, w$)
  $(q_x, y) \leftarrow (q_{0,\mathcal{H}}, w)$ ▷ output inconsistency to analyze in first iteration
  do
    $\langle \widehat{u}, \widehat{a}, \widehat{v} \rangle \leftarrow$ ANALYZE-OUTINCONS($|q_x|, y$) ▷ according to Lemma 3.8 (p. 42)
    SPLIT($\mathcal{H}, \mathcal{T}, \widehat{u}, \widehat{a}, \widehat{v}$) ▷ as in Algorithm 4.7, marking new inner node as temp.
    CLOSETRANSITIONS-SOFT($\mathcal{H}, \mathcal{T}$) ▷ cf. Algorithm 4.6
    while $\exists B \in \pi(\mathcal{T}), a \in \Sigma : \nexists B' \in \pi(\mathcal{T}) : \delta_{\mathcal{H}}(B, a) \subseteq B'$ **do** ▷ condition (5.1) violated
      ▷ finalization step
      $r_B \leftarrow blk\_root(B)$
      $succ\_lca \leftarrow lca_{\mathcal{T}}\{q.trans[a].tgt\_node \mid q \in B\}$ ▷ compute LCA of a-successors
      $v' \leftarrow succ\_lca.discriminator$
      $v \leftarrow a \cdot v$ ▷ assemble new final discriminator
      REPLACE-BLOCKROOT($\mathcal{T}, r_B, v$) ▷ cf. Algorithm 5.2
      CLOSETRANSITIONS-SOFT($\mathcal{H}, \mathcal{T}$)
    end while ▷ Postcondition: condition (5.1) holds
    if $\exists B \in \pi(\mathcal{T}) : |B| > 1$ **then** ▷ there are non-trivial blocks remaining
      ▷ condition (5.1) plus non-trivial blocks ⇒ output inconsistency
      $(q_x, y) \leftarrow \text{choose}\{(q, v) \mid q'_x \in B \wedge \exists o \in \mathbb{B} : o \neq \lambda^q_{\mathcal{H}}(v) \wedge (v, o) \in Sig_{\mathcal{T}}(q)\}$
      ▷ continue with analyzing chosen output inconsistency in next iteration
    end if
  while $\exists B \in \pi(\mathcal{T}) : |B| > 1$ ▷ Postcondition: all inner nodes are final
end procedure

observe that no more than $n-1$ temporary nodes will ever be added to the discrimination tree, and that each of the kn transitions is tested at most once against every temporary discriminator (this is due to the way in which the transition targets are preserved during subtree extraction, cf. Algorithm 5.3). Therefore, no more than in total $\mathcal{O}(kn^2)$ queries will every be performed during hard sifts.

Symbol Complexity. While hard sifts do not affect the asymptotic membership query complexity, the same cannot be said wrt. the symbol complexity. First, observe that the $\mathcal{O}(kn^2)$ queries during "regular" (i.e., neglecting the necessity for hard sifting) hypothesis construction contain $\mathcal{O}(kn^3)$ symbols, and that the number of symbols in queries for counterexample analysis is $\mathcal{O}(nm\log m)$, in accordance with Proposition 3.3. However, it may be necessary for counterexample analysis to sift every transition against (asymptotically) every temporary discriminator, which means that $\mathcal{O}(kn^2)$ queries, each containing $\mathcal{O}(n+m) = \mathcal{O}(m)$ symbols need to be performed. This results in an overall symbol complexity of $\mathcal{O}(kn^2m + nm\log m)$, which is the same as for Observation Pack. In Section 5.5, we will however see that the number is much smaller in practice.

Space Complexity. Storing the spanning-tree hypothesis requires space in $\Theta(kn)$. As remarked in Proposition 4.1, the discrimination tree—after elimination of all temporary discriminators, and assuming that final discriminators are stored in a trie—can be stored in linear space, i.e.,

$\Theta(n)$. Note that temporary discriminators are always suffixes of counterexamples, and thus can be stored in constant space (represented by a single index) in addition to the counterexample. Since the counterexample is provided to the learner from outside, i.e., the learner is not responsible for storing it, and all temporary discriminators have been eliminated when a refinement step is finished, it can be argued that all data under the control of the learner never requires more than $\Theta(kn)$ space.

The following proposition completes our preliminary analysis.

Proposition 5.1

The TTT *algorithm correctly infers a model for an unknown regular output function using at most $n-1$ equivalence queries and $\mathcal{O}(kn^2+n\log m)$ membership queries, which altogether contain $\mathcal{O}(kn^2m+nm\log m)$ symbols.*

5.3.2. Space Optimality

The loose analysis in the previous section showed that the overall space consumption is $\Theta(kn)$, and this space requirement is dominated by the spanning-tree hypothesis. In the introduction to this chapter, we have claimed that the space complexity of TTT is even optimal. While it may be intuitive that a "reasonable" automaton representation (e.g., via a transition table) needs $\Theta(kn)$ space, it is not entirely self-evident that there should not be *better* ways of storing canonical DFAs: after all, a transition table neither ensures that all states are reachable, nor that they are pairwisely inequivalent.

A first hint that this can be neglected is the (perhaps surprising) observation that the vast majority of all DFAs of a given size are canonical: according to Domaratzki *et al.* [63], there are 26,617,614 distinct (i.e., accepting distinct languages) canonical DFAs with $n=4$ states over an alphabet of size $k=3$. If the canonicity requirement is dropped (i.e., including also those languages that can be accepted by DFAs with $n\in\{1,2,3\}$ states), the number of distinct accepted *languages* grows by a mere 0.2% to 26,659,656.

However, these numbers only consider non-isomorphic DFAs, and it is still possible to encode the same (up to isomorphism) DFA in a transition table in $(n-1)!$ different ways. To *prove* optimality of the space complexity, we need to move away from a *uniform* cost model and analyze the space complexity in a *logarithmic* cost model, i.e., considering how many *bits* are required for the respective data structures.

It is well known that for encoding an object $x\in S$, on average $\lceil\log|S|\rceil$ bits are required to distinguish it from all the other objects in S. Thus, if the number of (non-isomorphic) canonical DFAs with n states over an alphabet of size k is $f_k(n)$, showing that the space complexity of TTT is in $\mathcal{O}(\log f_k(n))$ proves optimality.

Domaratzki *et al.* [63] give a lower bound of $f_k(n)\geq f_1(n)n^{(k-1)n}$, which can be intuitively explained as follows: let $\mathcal{A}$ be a canonical DFA over a unary alphabet Σ_1. If this DFA is extended to a DFA $\mathcal{A}'$ over $\Sigma\supseteq\Sigma_1$ by adding arbitrary transitions for input symbols in $\Sigma\setminus\Sigma_1$, $\mathcal{A}'$ remains canonical: every pair of states is separable by a word in Σ_1, and the Σ_1 transitions in $\mathcal{A}'$ are the same as in $\mathcal{A}$. Since there are $n^{(k-1)n}$ possible choices for the new transitions, the above result follows. Obtaining a lower bound for $f_1(n)$ is considerably harder, and we will content ourselves by simply reporting the combined result that $f_k(n)\sim n2^{n-1}n^{(k-1)n}$. As a consequence, we obtain $\log f_k(n)\in\Theta(kn\log n)$.

What about the space complexity of TTT in the logarithmic cost model? The discrimination tree contains at most $2n-1$ nodes, each of which needs to store a pointer to its parent.[9] Furthermore, inner nodes need to store pointers to their two children and to one of the $n-1$ nodes in the suffix trie, and leaves need to store a pointer to one of the n states in the hypothesis. As in all cases the number of potential target objects is in $\Theta(n)$, no more than $\Theta(\log n)$ bits are required per node in the discrimination tree, yielding an overall logarithmic space complexity of $\Theta(n \log n)$.

In the suffix trie, the only data that need to be stored for each node are its parent and the alphabet symbol labeling the outgoing edge. This yields a logarithmic space complexity of $\Theta(\log n + \log k)$ per node, and thus $\Theta(n \log n + n \log k)$ in total.

Again, the spanning tree hypothesis is the most crucial. Every state needs to maintain whether it is accepting or not ($\Theta(1)$ bits), a reference to its corresponding node in the discrimination tree ($\Theta(\log n)$ bits), its parent state in the spanning tree ($\Theta(\log n)$ bits), and k outgoing transitions. Each transition needs to store whether it is a tree or non-tree transition ($\Theta(1)$ bits), and its target state (tree transition) or node in the discrimination tree (non-tree transition), both of which can be referred to using $\Theta(\log n)$ bits. Note that it is not necessary to store the alphabet symbol associated with a transition, as this is given implicitly by the ordering of the outgoing transitions of a state (for a more efficient computation of access sequences, it makes sense to store the symbol associated with the unique incoming tree transition, which contributes an uncritical $\Theta(n \log k)$ bits in total). Combining all this, we obtain an overall space consumption of $\Theta(n + n \log k + n k \log n) = \Theta(k n \log n)$ bits for the spanning-tree hypothesis, which therefore also dominates the overall space consumption.

Proposition 5.2

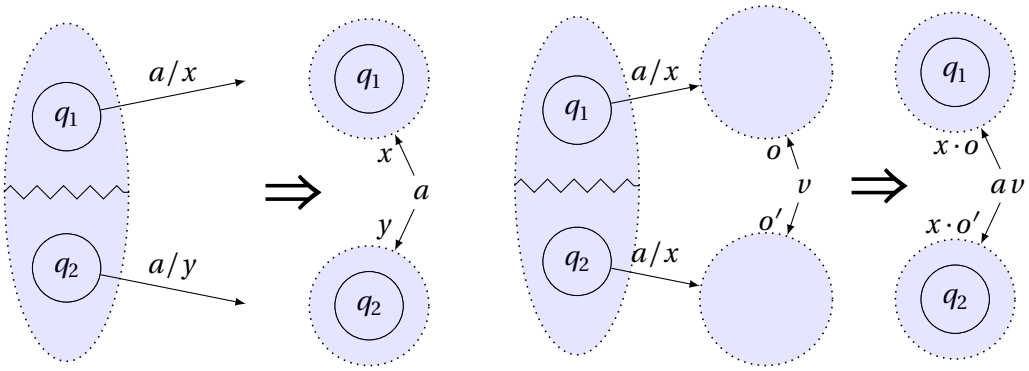
The TTT *algorithm is space-optimal, i.e., every correct active* DFA *learning algorithm has the same or a worse asymptotic space complexity in the logarithmic cost model.*

5.4. Adaptation for Mealy Machines

In this section, we briefly want to discuss how TTT can be adapted to learn Mealy machines. The formal framework has already been established in Section 3.5, including the necessary adaptions to data structures: for learning Mealy machines, the discrimination tree is no longer necessarily a binary tree, but inner nodes can have arbitrary outdegree. As a consequence, many new states can be discovered *en passant*, which CLOSETRANSITIONS from Algorithm 4.6 however already takes care of. Furthermore, the spanning-tree hypothesis must maintain for each transition its output symbol.

Finalization rules. Of particular importance for TTT is the finalization of discriminators. When learning Mealy machines, the separator for two states cannot always be extracted from the separator of its successors. Rather, two states might differ due to their transition outputs. Figure 5.12a illustrates the abstract finalization rule for this case: if the output of the a-transition

[9] In the algorithmic descriptions in this chapter, we have furthermore assumed that a node in the discrimination tree stores the set of all incoming non-tree transitions, which would require in total $\Theta\big(kn\log(kn)\big) = \Theta\big(kn(\log k + \log n)\big)$ additional bits, and thus exceed the established lower bound—at least for large alphabets, i.e., $k = n^{\omega(1)}$. However, these transitions can also be determined by iterating over all transitions in the entire hypothesis, and processing those that have a matching target node on-the-fly. This introduces additional computation effort, but does not affect correctness nor query and/or symbol complexities.



(a) Output rule

(b) Successor rule

Figure 5.12.: Abstract visualization of finalization rules for Mealy machines

of two states q_1 and q_2 differs, a can be used to separate these states, regardless of the targets of these transitions. Another adaption concerns the classical finalization rule, i.e., for states q_1, q_2, the a-transitions of which point into different blocks. In this case, the output of the a-transitions (say x, and assuming that the transitions have the same output, otherwise the aforementioned rule applies) can be prepended to the outputs of the a-successors wrt. their separator v (say, $o, o' \in \Omega^*$), to form the outputs separating q_1 and q_2 according to $a\,v$. This is illustrated in Figure 5.12b.

Output inconsistencies. If no finalization is possible while there still are non-singleton blocks, the TTT algorithm analyzes an output inconsistency (which must exist) to introduce a new state, as described in Section 5.2.3. An important observation was that the hypothesis, in spite of its non-determinism due to non-singleton blocks, nevertheless exposes a *deterministic* output behavior, i.e., condition (5.1) defined on p. 94 allowed us to define state output functions that are invariant under all possible resolutions of non-determinism (through hard sifting).

The finalization rule from Figure 5.12a ensures that, whenever no finalization step is possible, every two states in each block have the same transition outputs. Formally:

$$\forall B \in \pi(\mathcal{T}) : \forall q_1, q_2 \in B, a \in \Sigma : \gamma_{\mathcal{H}}(q_1, a) = \gamma_{\mathcal{H}}(q_2, a).$$

This means that it is possible to assign homogeneous transition outputs to entire *blocks*, which in conjunction with (5.1) can be extended from single symbols to words (i.e., the *extended* transition output function $\gamma^*_{\mathcal{H}}$ remains deterministic when lifted to sets of states within a single block). This makes it possible to define deterministic state output functions whenever no finalization is possible, thus allowing us to detect output inconsistencies.

Subtree extraction. A final modification concerns the extraction of subtrees, as described in Section 5.2.4 (in particular Algorithm 5.3). Since it is not known a priori which output values wrt. the replacement discriminator v will be observed (and the set of all possible outcomes $\Omega^{|v|}$ may be too large), the *inc* and *state* maps in Algorithm 5.2 have to be maintained as mapping nodes (or leaves, in the latter case) to (sparse) *maps* from Ω^* to sets of transitions or states. Consequently, the *mark* mapping maps nodes to subsets of Ω^*, and the subtree extraction has to be adapted to extract a subtree of every element of the *mark* set of the currently visited node (except for the case when the *mark* set contains only a single value, as then the inner node is

eliminated and the incoming transitions "fall through"). In particular, this means that lines 19–21 need to be adapted to call EXTRACT for every element of $mark[r_B]$, and the results of these calls form the children of the newly created inner node.

5.5. Evaluation

The query and symbol complexity analysis in Section 5.3 may seem a bit frustrating: all the additional effort did not allow us to reduce the asymptotic worst-case complexities of the Observation Pack algorithm. However, this analysis was based on overly pessimistic assumptions, such as the necessity for hard sifting *every* non-tree transition to enable counterexample analysis.

We therefore have conducted a series of experiments in which we attempt to measure the *practical* performance of active automata learning algorithms. As the results in this section will show, the TTT algorithm does in fact uniformly outperform every other algorithm in the presence of non-optimal counterexamples.

5.5.1. Evaluation Metrics

In accordance with the theoretical complexity analyses, we will measure the practical performance by considering how many *membership queries* were required to completely learn selected target systems, how many *symbols* these queries contain, and how many unsuccessful *equivalence queries* (i.e., how many counterexamples) were required.

Most learning algorithms (some more than others) pose the same queries more than once. These redundancies may be due to the inherent structure of the learning algorithm, or occur merely coincidentally, e.g., during counterexample analysis. In situations where membership queries are the predominant bottleneck, it is common to use a *cache* to store the answers to previously asked queries [130], avoiding duplicates. Thus, in some experimental setups, we will distinguish between *total queries* (those posed by the learner) and *unique queries* (those that could not be answered by the cache). The same applies to the number of symbols in these queries.

There are several possible ways of realizing equivalence queries. Since in all of our experiments we have a model of the target system at our disposal, it is possible to realize so-called "perfect" equivalence queries by means of simply checking equivalence between the hypothesis and the target DFA (e.g., using the near-linear algorithm by Hopcroft and Karp [89]). Such equivalence queries provide minimal counterexamples that are typically easy to analyze, but provide relatively little information.

In realistic scenarios, perfect equivalence queries are not available. Instead, equivalence queries are typically approximated using membership queries, often employing randomization (e.g., random sampling of words). Sophisticated strategies, which have proven to be quite successful in practice, have been presented by Howar [93]. However, the clever search for counterexamples is only indirectly related to the TTT algorithm itself (cf. also Section 4.2.4). Instead, we want to investigate the typical problem of such heuristics, namely that they often yield unnecessarily long counterexamples. Thus, exploiting our knowledge of the target systems in the experimental setup, we will randomly generate true counterexamples of certain lengths, and investigate the impact of the counterexample length on the performance.

We will not consider the actual (wall-clock) runtimes, as these primarily measure the quality of the implementation, and not of the algorithm itself. For this reason, the hardware specs of

Algorithm	Queries		Symbols		CEs
	total	unique	total	unique	
L*	2,294,747	2,232,100	28,311,774	27,605,073	74
Rivest/Schapire	1,508,780	1,464,614	17,728,161	17,257,794	70
Suffix1by1	1,551,177	1,503,369	18,275,210	17,758,490	68
Observation Pack	73,027	61,728	797,705	682,764	590
Kearns/Vazirani	102,615	61,554	1,088,394	681,213	592
TTT	72,361	61,535	793,465	681,528	592

Table 5.1.: Performance of selected learning algorithms on the `pots2` example ($n = 664$, $k = 32$) with perfect equivalence queries

the system on which the experiments were conducted do not matter. All experiments have been conducted on algorithms implemented in *LearnLib*[10] [112], a Java-based active automata learning framework developed by the author and others.

5.5.2. Realistic Systems

The first class of systems that we want to consider are models of "realistic" systems that were obtained from the CADP toolset [69]: a model of a *plain old telephony system* (`pots2`; $n = 664$, $k = 32$), and a model of Peterson's mutual exclusion protocol (`peterson3`; $n = 1328$, $k = 57$). These systems have frequently been used as benchmarks for active automata learning algorithms, e.g., by Berg *et al.* [32] and Howar [93].

Perfect Equivalence Queries

In the first series of experiments, we assume that perfect equivalence queries are available, i.e., the teacher provides minimal counterexamples to the learner. We compare the performance on the above-mentioned systems in this setting for six different algorithms:

- three observation-table based algorithms: the original L* by Angluin [19], the improved version by Rivest and Schapire [155], and the Suffix1by1 heuristic by Irfan *et al.* [105, 106]. Since the latter is guaranteed to only add a subset of the suffixes that would be added by L^*_{col} [127] and Shahbaz's algorithm [161], it is to be expected that these have a similar or worse performance than Suffix1by1.
- three discrimination tree-based algorithms: Kearns and Vazirani's algorithm [115], the Observation Pack by Howar [93], and the TTT algorithm.

The results for `pots2` are shown in Table 5.1. For the observation table-based algorithms, Rivest and Schapire's algorithm and Suffix1by1 perform slightly better than the original L* algorithm, and require roughly the same number of counterexamples. The discrimination tree-based algorithms all perform similar to each other when unique queries are considered (Kearns

[10] `http://learnlib.de/`

Algorithm	Queries		Symbols		CEs
	total	unique	total	unique	
L*	10,787,029	10,621,252	183,545,947	180,939,208	84
Rivest/Schapire	8,932,489	8,776,885	144,285,141	141,923,902	117
Suffix1by1	8,932,246	8,776,870	144,523,450	142,161,533	111
Observation Pack	149,021	123,420	2,114,674	1,754,539	1,202
Kearns/Vazirani	230,168	123,480	3,151,542	1,756,274	1,202
TTT	147,818	123,416	2,102,141	1,754,456	1,202

Table 5.2.: Performance of selected learning algorithms on the `peterson3` example ($n = 1328$, $k = 57$) with perfect equivalence queries

and Vazirani's algorithm poses significantly more duplicate queries than the other two algorithms). Their number of both queries and symbols is lower by a factor of 20–30x compared to the table-based algorithms, while requiring about ten times as many counterexamples.

Table 5.2, which displays the results for `peterson3`, shows similar characteristics: this time, the number of queries of the discrimination tree-based algorithms (which all three have almost the same performance, again with the notable exception that Kearns and Vazirani's algorithm poses more duplicate queries) is lower by a factor of 70–85x, and the number of symbols even by a factor of 80-100x. However, this time even more than ten times as many counterexamples are required.

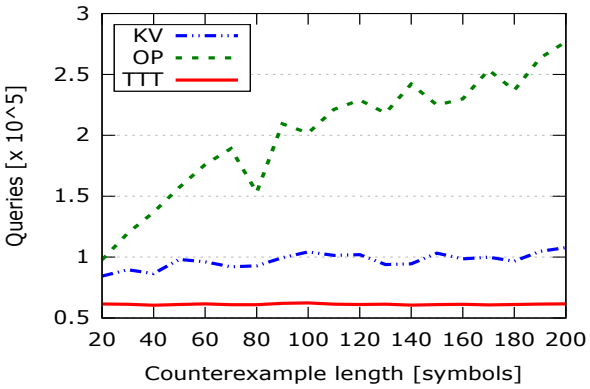
The vast performance difference between observation table-based and discrimination tree-based justifies to concentrate on the latter in the remaining evaluations. While they require significantly more equivalence queries, it is to be expected that exploiting the much lower number of membership queries to realize sophisticated heuristics for finding counterexamples [93, 94] is the much more beneficial approach in settings where equivalence queries can only be approximated (cf. also Section 4.2.4).

Counterexamples of Growing Length

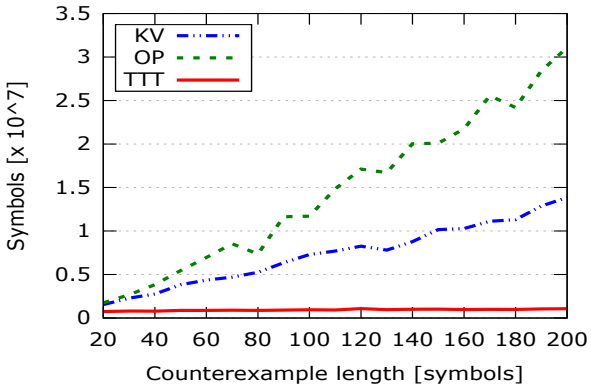
The previous setting of perfect counterexamples showed no significant differences between the three discrimination tree-based algorithms (if only unique queries are considered). This is not surprising: part of our motivation for the TTT algorithm was the problem of long counterexamples. Clearly, if counterexamples are of minimal length, the effort spent on replacing temporary discriminators can hardly yield significant returns (but also does not incur noticeable overhead).

For the next series of experiments, we change the setting to randomly generating true counterexamples of varying length (between 20 and 200, in increments of 10), and consider the (query and symbol) performance of a learning algorithm as a function of the counterexample length. Due to the observed characteristic of Kearns and Vazirani's algorithm to pose duplicate queries, a cache is employed also in this series of experiments, such that we will only consider *unique* queries. Furthermore, due to the findings reported by Isberner and Steffen [108] (cf. also Section 3.3.5), exponential search will be used to analyze counterexamples for all algorithms.[11]

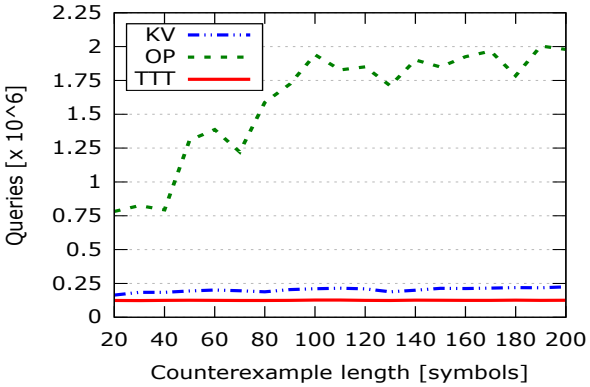
[11] Since Kearns and Vazirani's algorithm is prefix-based, while the other two algorithms are suffix-based, the direction of the search is reversed for the former.



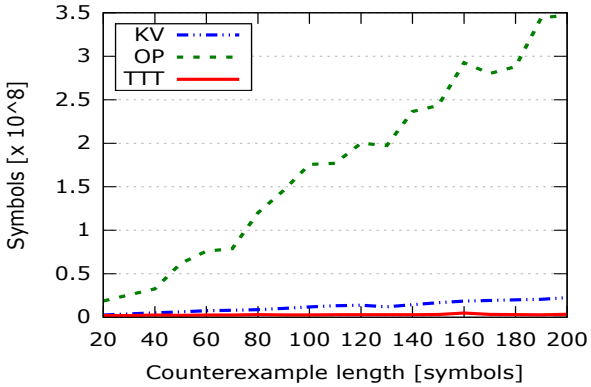
(a) Number of unique queries (`pots2`)



(b) Symbols in unique queries (`pots2`)

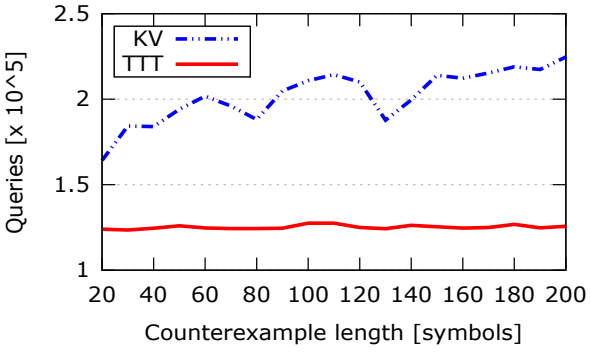


(c) Number of unique queries (`peterson3`)

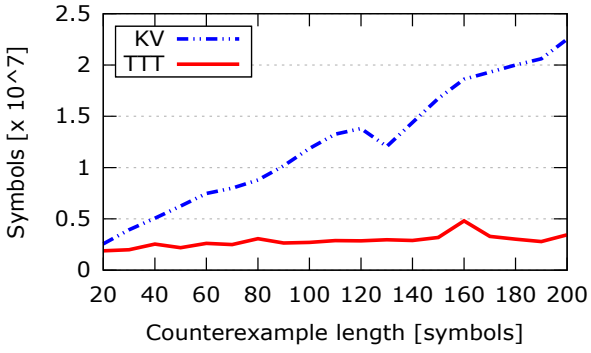


(d) Symbols in unique queries (`peterson3`)

Figure 5.13.: Performance of discrimination tree-based algorithms on `pots2` and `peterson3`

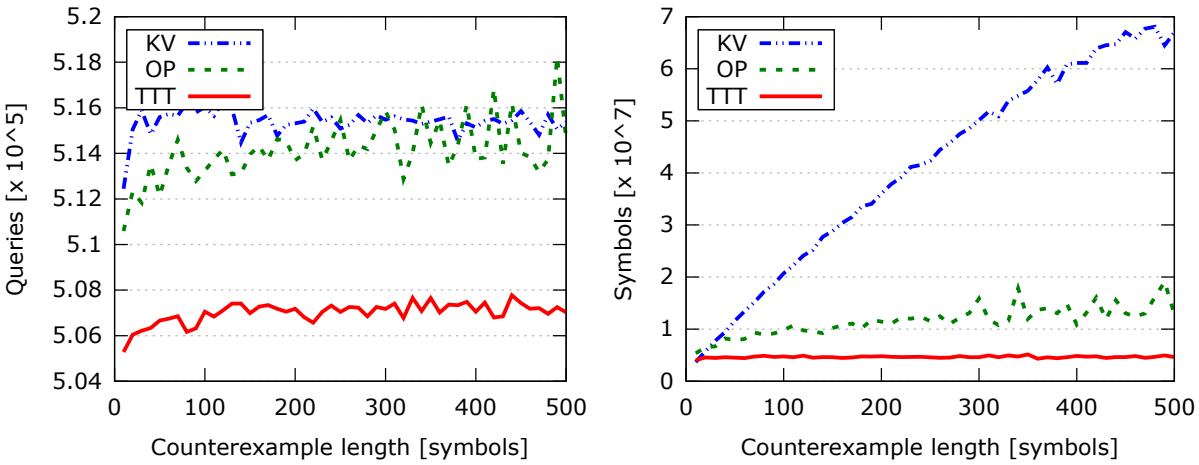


(a) Number of unique queries



(b) Number of symbols in unique queries

Figure 5.14.: Zoomed-in version of the plots from Figures 5.13c and 5.13d, excluding Observation Pack



(a) Number of unique queries

(b) Number of symbols in unique queries

Figure 5.15.: Results for a randomly generated DFA ($n = 1000$, $k = 50$)

The results for this series of experiments are shown in Figure 5.13. The Observation Pack algorithm (**OP**), on which TTT is based, performs rather poorly and is affected heavily by long counterexamples. Kearns and Vazirani's algorithm (**KV**) performs significantly better, but TTT still uses by far the least number of both queries and symbols. Even in the zoomed-in versions of the plots for `peterson3` (shown in Figure 5.14), hardly any impact of the counterexample length on TTT's performance can be seen.

5.5.3. Randomly Generated Automata

Randomly generated automata often exhibit characteristics which are rarely found in real-life systems. For example, they can typically be learned requiring only a small number of counterexamples even for discrimination tree-based algorithms. Howar *et al.* [94] report that in the ZULU challenge [58], in which the participants were to infer models of randomly-generated DFAs, on average three membership queries were sufficient to identify a new state. Nevertheless, the fact that randomly generating DFAs allows fine-tuning of their parameters (e.g., state space and alphabet size, ratio of accepting vs. non-accepting states etc.) makes them an important benchmark for evaluating learning algorithms.

Counterexamples of Growing Length

For the first series of experiments on randomly generated DFAs, a single DFA with $n = 1000$ states over an input alphabet of size $k = 50$ was generated.[12] The size of generated counterexamples was then increased from 10 to 500 in increments of 10, and for each counterexample length, 5 independent runs were conducted for each algorithm (on the same DFA, however).

The results are shown in Figure 5.15. TTT generally needs the lowest number of membership queries (cf. Figure 5.15a), but all three algorithms are within a very close range. Unlike in the

[12] The characteristics of the result did not change significantly when varying both the alphabet and the state space size.

previous series of experiments, the length of counterexamples seems to have no effect on the number of queries. A possible explanation is that for (uniformly) randomly generated systems, every word has the same expected discriminatory power, meaning that on average it partitions a randomly chosen set of states in two almost equal halves (the fact that the DFA itself was randomly generated prohibits any biases). This leads to nearly perfectly balanced discrimination trees. On systems that exhibit a more specific structure, long suffixes (as occurring when using the Observation Pack algorithm, but not Kearns and Vazirani's algorithm) are disadvantageous, as they are more specific and thus do not partition sets of states in a balanced way.

The combined number of symbols in all unique queries is shown in Figure 5.15b. Again, TTT remains nearly unaffected by counterexamples of growing length. However, in contrast to the previous series of experiments (cf. Figure 5.13), the performance of the Observation Pack algorithm is much closer to that of TTT, while the number of symbols required by Kearns and Vazirani's algorithm grows quickly with the length of the counterexamples.

Automata of Growing Size

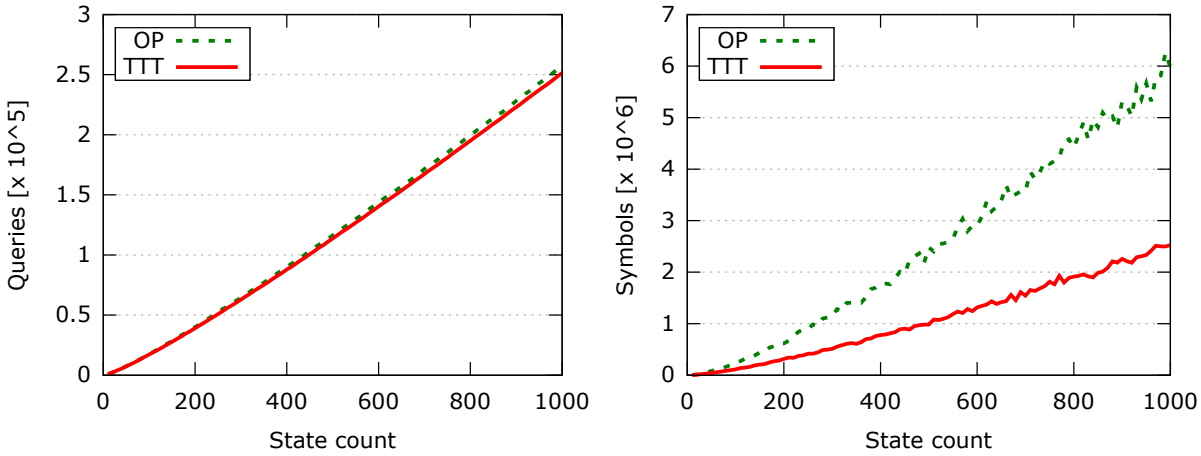
As the prefix-based counterexample analysis strategy implemented in Kearns and Vazirani's algorithm handles long counterexamples on randomly generated DFAs rather poorly, in our last series experiments we focus on a direct comparison between Observation Pack and TTT. This time, we randomly generated DFAs of sizes between 10 and 1000 (in increments of 10) over an alphabet of size $k = 25$, and averaged both membership queries and symbol counts over 10 runs on different automata with the same state count.

For the previous experiment series with non-optimal counterexamples, we used a fixed counterexample length throughout the entire course of a single learning process. This time, we choose as the length for each counterexample 1.5 times the number of states in the current hypothesis. Note that this means that counterexamples in early phases of the learning process will be shorter, which should not make a difference for TTT but might be advantageous for other algorithms.

The number of queries, shown in Figure 5.16a, shows hardly any difference between TTT and Observation Pack. In fact, both curves almost perfectly fit $kn\log n$, which is the "optimal" query complexity with a fully balanced discrimination tree. Still, the difference in the number of symbols (cf. Figure 5.16b) is significant: TTT requires only about one-third as many symbols as Observation Pack, even though the generated counterexamples were of rather moderate length.

5.5.4. Interpretation of the Results

The experimental results we reported on in the previous subsection clearly position TTT as the preferable learning algorithm in almost any circumstance: in the settings with perfect equivalence queries, when we can expect nothing to be gained by optimizations geared towards excessively long counterexamples, TTT is on par with the other discrimination tree-based algorithms (which, in turn, are clearly superior to the observation table-based ones). However, in virtually every scenario with non-minimal counterexamples, TTT requires a significantly lower number of symbols (and sometimes also queries) than the other two discrimination tree-based algorithms. In fact, TTT seems to be virtually unaffected by growing counterexample lengths, as neither its symbol nor query performance changes observably.



(a) Number of unique queries

(b) Number of symbols in unique queries

Figure 5.16.: Results for randomly generated DFAs of growing size ($k = 25$), using $1.5|\mathcal{H}|$ as counterexample length

An interesting aspect is that in some settings, the Observation Pack algorithm clearly beats Kearns and Vazirani's algorithm, while in other settings it is the other way round. This is most likely due to different characteristics of prefix- and suffix-based counterexample analysis, which constitutes the main difference between the two algorithms: apparently, either the structure of the system or the method of generating counterexamples in one case favors prefix-based analyses, whereas in the other case it favors suffix-based analyses. The TTT algorithm, on the other hand, always outperforms them both, even in settings where the suffix-based analysis which it is based upon is apparently disadvantaged.

Moreover, an important observation is that on systems of realistic structure, TTT significantly reduces the number of membership queries compared to the other two algorithms, while this effect cannot be observed for randomly generated systems. We conjecture that this is due to the fact that in the former case, the shorter discriminators found by TTT partition the sets of states more evenly, leading to better-balanced discrimination trees. This also explains the advantage of Kearns and Vaziranis' algorithm over Observation Pack on these systems, and furthermore once again shows that randomly generated automata have their limits when it comes to exposing characteristics of learning algorithms.

It should be noted that in all experiments, the length of the counterexamples were rather moderate, as they hardly exceeded a length of n. If counterexamples result from monitoring executions of live systems, as sketched in Figure 5.1, traces can easily exceed lengths of tens of thousands of symbols. It can be clearly stated that in such a setting, the use of any other algorithm than TTT is simply infeasible, as the plots clearly show that the gap between TTT's performance and that of the other algorithms grows further—even in relative terms—with increasing counterexample lengths.

6. Learning Visibly Pushdown Automata

In the previous part of this thesis, we have examined the foundation of learning finite-state machines. In particular, we have developed a framework that allows to treat the majority of existing learning algorithms uniformly, which is essential for identifying and comparing their characteristics. These considerations have led to the identification of a number of *desirable* properties, and, based on these, the development of an algorithm with superior practical performance.

However, the approach laid out in the previous chapters inherently relies on approximating—and finally identifying—the Nerode congruence. Thus, it is constrained to $\mathscr{L}_3$, i.e., the class of regular languages, as languages in these class are characterized precisely by their number of Nerode equivalence classes being finite. The finite index of the Nerode congruence guarantees that, after a finite number of refinement steps (counterexamples), we obtain a correct model for the target language (cf. Theorem 3.2, p. 33).

The above restriction hinders practical applications of active automata learning, as many real-life systems exhibit non-regular behavior, i.e., they cannot be modeled using finite-state machines, as they maintain some form of unbounded memory.[1] A possible approach is to switch the goal from *identifying* a certain language (or, more generally, output function) to merely *approximating* it. As a non-regular language is characterized by the fact that recognizing it requires unbounded space, it can be approximated by placing a finite bound on this space requirement. As an intuitive example, it is well known that the language of matching parentheses (i.e., $L_() = \{\varepsilon, (), ()(), (()), \ldots\}$) is not regular. However, if a bound is placed on the nesting depth of parentheses, it becomes regular again. Other non-regular languages can be approximated in a similar fashion.

If the methods presented so far are applied to non-regular languages, an equivalence query would, in principle, never indicate success, but continuously provide new counterexamples. These counterexamples determine the nature of the approximation. The extent to which the result *preserves* the semantics of the original language is highly dependent on the counterexamples provided to the learner, a factor that cannot always be controlled. Moreover, there might be languages that are inherently irregular to such an extent that no reasonable regular approximation conveys their essence in a satisfactory way.

The question of how much further up in the Chomsky hierarchy we can go and still obtain results as strong as for the regular case naturally arises. Unfortunately, we already hit a road block when considering the next class $\mathscr{L}_2$ in the Chomsky hierarchy, i.e., the class of context-free languages. While having been actively investigated, learning formal descriptions of context-free languages (such as context-free grammars) comes with numerous difficulties, and theoretical learnability results for the full class are generally negative (de la Higuera [61] provides a survey),

[1] There typically are two forms of manifestations of this unboundedness (which can occur simultaneously): in one case, there is a *finite* number of memory locations, which however can store data values from an *infinite* domain (e.g., $\mathbb{N}$). An example for this class are register automata [43, 114]. Another model is to assume an *unbounded* number of memory locations, each of which can store values from a *finite* domain. An example for the latter are the visibly pushdown automata that we will consider in this chapter.

or overly restricted: Angluin [19], for instance, discusses active learning of context-free grammars in Chomsky normal form, but under the assumption that the set of non-terminals and the start symbol are known to the learner, and furthermore membership queries can be asked for each language generated by a specified non-terminal, not just the start symbol. On the more fundamental side, even in a white-box setting it is impossible to realize a *minimally adequate teacher*, as the equivalence of two context-free grammars is undecidable.

It is well known that the machine model corresponding to context-free languages are pushdown automata (PDA). A PDA can be described as a (nondeterministic) finite automaton equipped with a stack. Symbols (from a finite alphabet, which may be entirely different from the input alphabet) can be pushed onto the stack, and the behavior of the automaton may depend on the symbol on top of the stack, which can be removed (popped) during the execution of a transition.

The fact that PDAs are equipped with a stack makes them attractive as a model for systems with function calls and recursion [41, 42], which can also be modeled by pushing the current state onto the stack when a call is made, and restoring the state upon returning by popping the old state from the stack. Unfortunately, PDAs in general are a far too strong model for this application in a verification context: from universality over equivalence to inclusion, pretty much all properties concerning unrestricted context-free languages are undecidable [78].

The strict subclass of *deterministic context-free languages* [72], which can be recognized by deterministic pushdown automata (DPDA), is significantly more well-behaved in this regard: equivalence and universality are decidable for DPDAs, but other properties such as inclusion remain undecidable [23, 67]. Moreover, the class of deterministic context-free languages lacks a number of closure properties: the union of two deterministic context-free languages might be a nondeterministic context-free language. To summarize, even if we were able to obtain PDA or DPDA models by active learning, they would most likely be of only limited use in many application contexts due to their computational intractability.

As a remedy, Alur and Madhusudan [11] proposed *visibly pushdown languages* (VPLs) as a restricted form of context-free languages that admit decidability of the majority of interesting properties, and are closed under most operations such as complementation, union, or intersection. Thus, VPLs mirror many desirable characteristics of regular languages, even though the complexities of most operations are much higher than for regular languages [166]. The corresponding machine model are *visibly pushdown automata* (VPAs), which constitute a restricted form of PDAs.

The word "visibly" refers to the fact that each symbol of the input alphabet (usually denoted by $\widehat{\Sigma}$, see below for a formal description) belongs to exactly one of three classes, and each class uniquely determines the stack operation: *call symbols* push a symbol onto the stack, while *return symbols* pop a symbol off the stack; *internal symbols* do not modify or even inspect the stack. It should be noted that the restriction compared to general (deterministic) context-free languages manifests itself not primarily in the fact that it is communicated to the outside what actions are performed on the stack (and the symbols being pushed onto the stack furthermore remain invisible to the outside), but rather in the fact that there is a fixed association between input symbols and stack actions. For example, the language $L_{()}$ of matched parentheses is a VPL for a suitable partition of the input alphabet (i.e., if "(" is treated as a call and ")" as a return symbol), while $L_{Pal} = \left\{ w\, w^{\mathrm{R}} \mid w \in \Sigma^* \right\}$, the language of even-length palindromes, is not a VPL, as each symbol would need to behave as a call symbol in the first half, and as a return symbol in the

second half. However, if VPAs are used as a model for programs with recursion, this restriction is negligible, as calls and returns are both visible and clearly designated.

Another interesting property of VPLs (which they share with regular languages) is that there is no loss in expressive power when constraining the corresponding machine model (finite automata or VPAs) to deterministic behavior. That is, any VPL can be recognized by a deterministic VPA, and any non-deterministic VPA can be *determinized* without changing the accepted language. While this possibly incurs an exponential blow-up, it allows us to focus on the conceptually simpler deterministic version without any loss of expressive power. Thus, in the sequel we generally write VPA to refer to the deterministic version, unless otherwise noted.

The favorable properties of VPLs and VPAs make them a natural candidate for investigating the extent to which the framework developed for actively learning regular languages can be transferred to a richer class. Kumar *et al.* [119] have presented a learning algorithm for a special type of VPAs, called *modular* VPA*s*. Their learning algorithm is an adaption of the algorithm by Kearns and Vazirani [115], and while their notion of modular VPAs is restricted in the sense that there can only be a single return, it can easily be generalized. However, their algorithm contains no optimizations whatsoever, and in particular may have an exponential query complexity even if only minimal counterexamples are provided. In this chapter, we will show how the techniques from the previous chapters, which paved the way for an efficient DFA learning algorithm, can be transferred to learning VPAs.

6.1. Preliminaries

We start by formalizing some of the above-mentioned concepts. The first step is to define the adapted alphabet structure, with its designated call and return symbols. The majority of definitions and propositions in this section can be found similarly in the original paper describing VPLs by Alur and Madhusudan [11], some of which have been adjusted to make for a simpler presentation of the main contents of this chapter.

Definition 6.1 (*Visibly Pushdown Alphabet*)

A *visibly pushdown alphabet* is a triple $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$, where

- Σ_{call} is a finite set of *call symbols*,
- Σ_{ret} is a finite set of *return symbols*,
- Σ_{int} is a finite set of *internal symbols*,

and $\Sigma_{call}, \Sigma_{ret}, \Sigma_{int}$ are pairwisely disjoint.

In the sequel, we will identify $\widehat{\Sigma}$ with the set $\Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{int}$, i.e., the disjoint union of all its component sets. This allows us to write $\widehat{\Sigma}^*$ to denote the set of all words over $\widehat{\Sigma}$.

6.1.1. Well-Matched Words

As mentioned in the introduction of this chapter, *call* and *return* symbols correspond to *push* and *pop* operations on the stack of a recognizing VPA. The set $\widehat{\Sigma}^*$ contains all possible (finite) sequences over $\widehat{\Sigma}$, including those that begin with a return symbol (corresponding to a pop on

an empty stack), and those that end with a call symbol (corresponding to a non-empty stack at the end). While it is perfectly possible to define corresponding semantics for these cases, it often makes sense (and considerably simplifies presentation) to explicitly exclude such cases. The following definition helps formalizing this.

Definition 6.2 (*Call/return balance*)

Let $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$ be a visibly pushdown alphabet. The *call/return balance* is a function $\beta : \widehat{\Sigma}^* \to \mathbb{Z}$, defined as

$$\beta(\varepsilon) =_{df} 0, \quad \beta(w \cdot a) =_{df} \beta(w) + \begin{cases} 1 & \text{if } a \in \Sigma_{call} \\ -1 & \text{if } a \in \Sigma_{ret} \\ 0 & \text{if } a \in \Sigma_{int} \end{cases} \quad \forall w \in \widehat{\Sigma}^*, a \in \widehat{\Sigma}.$$

Note that the call/return balance β is a purely syntactical measure: for instance, it does not depend on any semantical assumptions about what popping an empty stack entails. However, it allows us to concisely define the concept of *call-matched*, *return-matched*, and *well-matched* words.

Definition 6.3 (*Call-matched, return-matched, well-matched*)

Let $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$ be a visibly pushdown alphabet. $w \in \widehat{\Sigma}^*$ is called ...

(i) *return-matched* if and only if for all prefixes $u \in \mathrm{Pref}(w)$, we have $\beta(u) \geq 0$. The set of return-matched words over $\widehat{\Sigma}$ is denoted by $\mathrm{MR}(\widehat{\Sigma})$.

(ii) *call-matched* if and only if for all suffixes $v \in \mathrm{Suff}(w)$, we have $\beta(v) \leq 0$. The set of call-matched words over $\widehat{\Sigma}$ is denoted by $\mathrm{MC}(\widehat{\Sigma})$.

(iii) *well-matched* if and only if w is both return-matched and call-matched. The set of well-matched words over $\widehat{\Sigma}$ is denoted by $\mathrm{WM}(\widehat{\Sigma}) = \mathrm{MR}(\widehat{\Sigma}) \cap \mathrm{MC}(\widehat{\Sigma})$.

The following properties of call-, return-, and well-matched words complement the above, rather technical definition by providing an intuition on how these words are structured.

For any word $w \in \widehat{\Sigma}^*$ of the form $w = u\,c\,v$ $(u, v \in \widehat{\Sigma}^*, c \in \Sigma_{call})$, if there exists a decomposition of v into $v = v'\,r\,v''$ $(v', v'' \in \widehat{\Sigma}^*, r \in \Sigma_{ret})$ such that v' is well-matched, we call r the *matching return* for c.[2] If such a decomposition exists, it is unique, and we say that c is *matched* (or is a *matched call*) in w. Otherwise, we say that c is *unmatched* (in w).

Analogously, if $w = u\,r\,v$ $(u, v \in \widehat{\Sigma}^*, r \in \Sigma_{ret})$, we say that r is matched (or is a *matched return*) if there exists a decomposition of u into $u'\,c\,u''$ such that u'' is well-matched, and c is called the *matching call* for r. Again, this decomposition is unique if it exists, and if it does not exist, we say that r is unmatched (in w). We write $w = u\,\underbrace{c\,w'\,r}\,v$ to express the fact that c and r match each other in w.

If $w \in \mathrm{MR}(\widehat{\Sigma})$, every return symbol in w is matched, and we can uniquely decompose w into $w = w_1 c_1 w_2 c_2 \ldots c_{m-1} w_m$, such that each w_i, $1 \leq i \leq m$, is a well-matched word, and $c_i \in \Sigma_{call}$, $1 \leq i < m$, are the unmatched calls in w. The same works for a word $w \in \mathrm{MC}(\widehat{\Sigma})$, which can be

[2] Here, the identifiers c and r refer to the symbols in the context of the word w, not symbols from Σ_{call} and Σ_{ret} as isolated entities.

decomposed into $w_1 r_1 w_2 r_2 \dots r_{m-1} w_m$, where each w_i is well-matched, and $r_i \in \Sigma_{ret}$, $1 \leq i < m$, are the unmatched returns in w. Finally, a well-matched word $w \in \mathrm{WM}(\widehat{\Sigma})$ contains neither unmatched calls nor unmatched returns, which implies that each call symbol in w can be uniquely associated with a return symbol in w, and vice versa.

6.1.2. Visibly Pushdown Automata

As mentioned before, a VPA is basically a finite-state machine equipped with an (unbounded) stack, which can store symbols ranging over some alphabet Γ. Formally, we can model the contents of a stack as a word over Γ, where the first symbol corresponds to the topmost symbol on the stack, and so on. The empty stack is represented by the empty word ε. We define three functions for operating on a stack: *push* adds a symbol to the top of the stack, while *pop* removes the topmost symbol (it has no effect when invoked on the empty stack). Finally, *peek* returns the top of a non-empty stack, and returns a special symbol $\perp$ (that is not part of the stack alphabet Γ) when invoked on the empty stack. Note that in the context of VPAs, *peek* always occurs in conjunction with *pop*; merely inspecting the topmost symbol without removing it is not possible.

These operations can be formally defined as follows:

$$\begin{array}{lll} push\colon \Gamma^* \times \Gamma \to \Gamma^*, & push(\sigma,\gamma) =_{df} \gamma\cdot\sigma & \forall \sigma \in \Gamma^*, \gamma \in \Gamma \\ peek\colon \Gamma^* \to \Gamma \cup \{\perp\}, & peek(\varepsilon) =_{df} \perp,\ peek(\gamma\cdot\sigma) =_{df} \gamma & \forall \sigma \in \Gamma^*, \gamma \in \Gamma \\ pop\colon \Gamma^* \to \Gamma^* & pop(\varepsilon) =_{df} \varepsilon,\ pop(\gamma\cdot\sigma) =_{df} \sigma & \forall \sigma \in \Gamma^*, \gamma \in \Gamma \end{array}$$

Definition 6.4 (*VPDA*)

Let $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$ be a visibly pushdown alphabet. A (deterministic) *visibly pushdown automaton* (VPA) over $\widehat{\Sigma}$ is a tuple $\mathcal{A} = \left\langle L_\mathcal{A}, \widehat{\Sigma}, \ell_{0,\mathcal{A}}, \Gamma_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A} \right\rangle$, where

- $L_\mathcal{A}$ is a finite, non-empty set of *locations*,
- $\ell_{0,\mathcal{A}} \in L_\mathcal{A}$ is the *initial location*,
- $\Gamma_\mathcal{A}$ is the *stack alphabet*,
- $\delta_\mathcal{A}$ is the *transition function*, and is defined as the union of three functions $\delta_\mathcal{A} = \delta_{call,\mathcal{A}} \cup \delta_{ret,\mathcal{A}} \cup \delta_{int,\mathcal{A}}$, where
  - $\delta_{call,\mathcal{A}}\colon L_\mathcal{A} \times \Sigma_{call} \to L_\mathcal{A} \times \Gamma_\mathcal{A}$ is the *call transition function*,
  - $\delta_{ret,\mathcal{A}}\colon L_\mathcal{A} \times \Sigma_{ret} \times (\Gamma_\mathcal{A} \cup \{\perp\}) \to L_\mathcal{A}$ is the *return transition function*,
  - $\delta_{int,\mathcal{A}}\colon L_\mathcal{A} \times \Sigma_{int} \to L_\mathcal{A}$ is the *internal transition function*, and
- $F_\mathcal{A} \subseteq L_\mathcal{A}$ is a set of *accepting* (or *final*) locations.

Semantics of a VPA. We describe the semantics of a VPA $\mathcal{A}$ in terms of an infinite-state transition system, where $\mathcal{S} =_{df} L_\mathcal{A} \times \Gamma_\mathcal{A}^*$ is the state (or *configuration*) space, and $Act =_{df} \widehat{\Sigma}$ defines the set of actions. The initial configuration is $\left\langle \ell_{0,\mathcal{A}}, \varepsilon \right\rangle$. The transition relation $\longrightarrow \subseteq (L_\mathcal{A} \times \Gamma_\mathcal{A}^*) \times \widehat{\Sigma} \times (L_\mathcal{A} \times \Gamma_\mathcal{A}^*)$ is defined as follows:

- $\langle \ell, \sigma \rangle \xrightarrow{i} \left\langle \ell', \sigma \right\rangle$ if and only if $\delta_{int,\mathcal{A}}(\ell, i) = \ell'$ (for all $\ell, \ell' \in L_\mathcal{A}, i \in \Sigma_{int}, \sigma \in \Gamma_\mathcal{A}^*$)

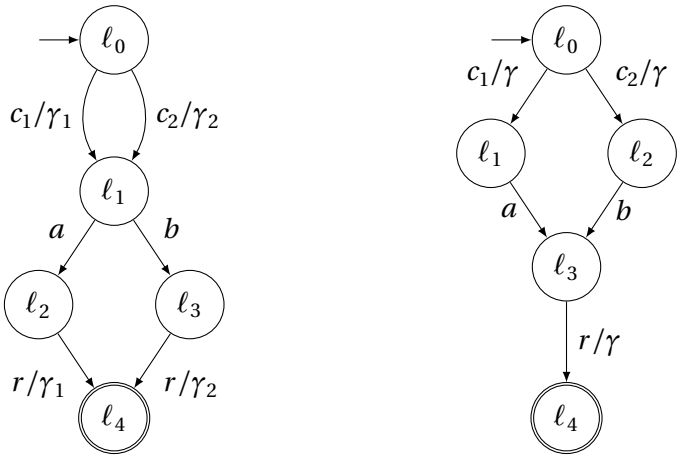


Figure 6.1.: Two VPAs accepting the same language $\mathcal{L} = \{c_1\, a\, r, c_2\, b\, r\}$

- $\langle \ell, \sigma \rangle \xrightarrow{c} \langle \ell', push(\sigma, \gamma) \rangle$ if and only if $\delta_{call,\mathcal{A}}(\ell, c) = (\ell', \gamma)$ (for all $\ell, \ell' \in L_\mathcal{A}, c \in \Sigma_{call}, \gamma \in \Gamma, \sigma \in \Gamma_\mathcal{A}^*$)
- $\langle \ell, \sigma \rangle \xrightarrow{r} \langle \ell', pop(\sigma) \rangle$ if and only if $\delta_{ret,\mathcal{A}}(\ell, r, peek(\sigma)) = \ell'$ (for all $\ell, \ell' \in L_\mathcal{A}, r \in \Sigma_{ret}, \sigma \in \Gamma_\mathcal{A}^*$)

We define the (extended) transition function $\delta_\mathcal{A}$ to reason about successor states using functional notation, i.e., we treat it as a function $\delta_\mathcal{A} : (L_\mathcal{A} \times \Gamma_\mathcal{A}^*) \times \widehat{\Sigma}^* \to (L_\mathcal{A} \times \Gamma_\mathcal{A}^*)$, where the value of $\delta_\mathcal{A}(\langle \ell, \sigma \rangle, w)$ is defined as the unique pair $\langle \ell', \sigma' \rangle \in L_\mathcal{A} \times \Gamma_\mathcal{A}^*$ satisfying $\langle \ell, \sigma \rangle \overset{w}{\Rightarrow} \langle \ell', \sigma' \rangle$. A word $w \in \widehat{\Sigma}^*$ is *accepted* by a VPA if and only if there exist $\ell \in F_\mathcal{A}, \sigma \in \Gamma_\mathcal{A}^*$ such that $\langle \ell_{0,\mathcal{A}}, \varepsilon \rangle \overset{w}{\Rightarrow} \langle \ell, \sigma \rangle$. The *language* of a VPA is defined as the set of all words it accepts, i.e.,

$$\mathcal{L}(\mathcal{A}) =_{df} \left\{ w \in \widehat{\Sigma}^* \mid \exists \ell \in F_\mathcal{A}, \sigma \in \Gamma_\mathcal{A}^* : \langle \ell_{0,\mathcal{A}}, \varepsilon \rangle \overset{w}{\Rightarrow} \langle \ell, \sigma \rangle \right\}.$$

Any language $\mathcal{L} \subseteq \widehat{\Sigma}^*$ that is accepted by some VPA is a *visibly pushdown language* (VPL).

Visualization of VPAs. VPAs are visualized in a manner similar to finite-state machines, i.e., as a graph representing the transition structure. Two example VPAs over the visibly pushdown alphabet $\widehat{\Sigma} = \langle \Sigma_{call}, \Sigma_{ret}, \Sigma_{int} \rangle$, where $\Sigma_{call} = \{c_1, c_2\}$, $\Sigma_{ret} = \{r\}$, and $\Sigma_{int} = \{a, b\}$, and accepting the language $\mathcal{L} = \{c_1\, a\, r, c_2\, b\, r\}$, are shown in Figure 6.1: locations are drawn as circles, and an incoming arrow without a source node indicates the initial location. Accepting locations are drawn as double circles. Internal transitions are simply labeled with the respective internal action. Labels of call transitions are of the form c/γ, where $c \in \Sigma_{call}$ is the call action, and $\gamma \in \Gamma$ is the stack symbol that is being *pushed* onto the stack. Labels of return transitions look similarly, i.e., they are of the form r/γ for a return symbol $r \in \Sigma_{ret}$, but in this context $\gamma \in \Gamma$ is the stack symbol that is being *popped* from the stack. Not all transitions are shown in the VPAs from Figure 6.1; those that are omitted lead into a sink location (not shown).

Restriction to well-matched words. Note that an empty stack is no prerequisite for acceptance, and imposing this strictly reduces the expressive power, as the contents of the stack are controlled by the call and return symbols occurring in a word $w \in \widehat{\Sigma}^*$ only (in other words, this constrains the acceptable languages to subsets of $\mathrm{MC}(\widehat{\Sigma})$). Similarly, encountering a return symbol

when the stack is empty does not necessarily result in rejection, and enforcing this constrains the acceptable languages to subsets of $\mathrm{MR}(\widehat{\Sigma})$. Thus, if both an empty stack is defined as a prerequisite for acceptance, and return transitions on $\bot$ are prohibited, the accepted language is necessarily *well-matched*. It is common to superimpose these rules by considering the *well-matched language* $\mathcal{L}_{\mathrm{WM}}(\mathcal{A}) =_{df} \mathcal{L}(\mathcal{A}) \cap \mathrm{WM}(\widehat{\Sigma})$ of $\mathcal{A}$. Since $\mathrm{WM}(\widehat{\Sigma})$ is a VPL, and VPLs are closed under intersection, $\mathcal{L}_{\mathrm{WM}}(\mathcal{A})$ is always a VPL. In the sequel, we will only consider well-matched languages, as it greatly simplifies the presentation.

For a well-matched word $w \in \mathrm{WM}(\widehat{\Sigma})$, we always have $\delta_{\mathcal{A}}(\langle \ell_{0,\mathcal{A}}, \varepsilon \rangle, w) = \langle \ell, \varepsilon \rangle$ for some $\ell \in L_{\mathcal{A}}$. This gives rise to define the *location* reached in $\mathcal{A}$ by a well-matched word $w \in \mathrm{WM}(\widehat{\Sigma})$, denoted by $\mathcal{A}[w]$, as the location ℓ making the above equation true.

Remark 6.1

In particular in the context of automata learning, it is much more convenient to consider *output functions* instead of languages. As we constrain ourselves to well-matched languages only, we generally assume output functions to be of the form $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$, instead of the more general signature $\lambda : \widehat{\Sigma}^* \to \mathbb{B}$, and refer to such a λ as a *well-matched output function*. Expressions of the form $\lambda(w)$, where $w \in \widehat{\Sigma}^* \setminus \mathrm{WM}(\widehat{\Sigma})$, are still permitted, but assumed to be 0 regardless of the concrete function λ.

For a VPA $\mathcal{A}$, the (well-matched) output function $\lambda_{\mathcal{A}} : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ is defined as the characteristic function of $\mathcal{L}_{\mathrm{WM}}(\mathcal{A}) \subseteq \mathrm{WM}(\widehat{\Sigma})$. If for some well-matched output function $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ there exists a VPA $\mathcal{A}$ satisfying $\lambda = \lambda_{\mathcal{A}}$, we refer to $\mathcal{A}$ as a *(well-matched) visibly pushdown (language) output function*, or simply a *(well-matched)* VPL *output function*.

6.1.3. 1-SEVPAs and Normalized Stack Alphabets

Alur *et al.* [14] have shown that, in general, there is no unique minimal VPA for a given VPL $\mathcal{L} \subseteq \widehat{\Sigma}^*$. The reason is the degree of freedom permitting to distribute information across both locations and stack contents. Consider, for example the VPAs from Figure 6.1. Both have the same number of locations, without being isomorphic to each other. Furthermore, merging any two locations inevitably changes the accepted language (note that for these VPAs, we did not assume a restriction to well-matched words). However, the VPA shown in Figure 6.1a, while in the initial location "remembers" the call symbol (c_1 or c_2, determining whether a or b is expected subsequently) by pushing either γ_1 or γ_2 onto the stack, and moving to the target location ℓ_1. In contrast, the VPA from Figure 6.1b pushes the same stack symbol γ onto the stack for both call symbols, but remembers the call symbol by its choice of the target location (ℓ_1 or ℓ_2).

This conflict can only be resolved by imposing some restriction on the form of a VPA. In the model of *k-module single entry visibly pushdown automata* (k-SEVPAs), this is achieved by partitioning Σ_{call} to form a partition $\{\Sigma_{call}^j\}_{j=1}^k$ of size k,[3] and requiring that for each $c \in \Sigma_{call}$, the successor location be solely defined by the partition class to which c belongs. The set of locations is furthermore partitioned into $k+1$ subsets ("modules"), such that internal transitions only run within the same module, and there is only one entry location (i.e., call target) per module. It can be shown that, for a fixed k-partition of Σ_{call}, $1 \leq k \leq |\Sigma_{call}|$, there is a unique (up to isomorphism) minimal (i.e., *canonical*) k-SEVPA for every well-matched VPL.

[3] Note that k here is *not* the size of the alphabet.

We will focus on 1-SEVPA here, which do not require choosing a partition of the call alphabet, and are characterized by the restriction that the target location of every call transition in a 1-SEVPA $\mathcal{A}$ is the initial location $\ell_{0,\mathcal{A}}$.[4]

Another simplification can be achieved by *normalizing* the stack alphabet. In a complete VPA $\mathcal{A}$ with location set $L_{\mathcal{A}}$, there are exactly $|L_{\mathcal{A}}| \cdot |\Sigma_{call}|$ call transition. Thus, changing the stack alphabet to $L_{\mathcal{A}} \times \Sigma_{call}$, and requiring that every call transition in $\mathcal{A}$ be of the form $\delta_{call,\mathcal{A}}(\ell, c) = (\ell', (\ell, c))$ for all $\ell \in L_{\mathcal{A}}, c \in \Sigma_{call}$ and some $\ell' \in L_{\mathcal{A}}$ (and replacing the stack symbols in the corresponding return transitions with their normalized form) does not change the semantics, as it corresponds to a stack alphabet of the finest possible granularity. In conjunction with the 1-SEVPA property, every call transition will thus be of the form $\delta_{call,\mathcal{A}}(\ell, c) = \big(\ell_{0,\mathcal{A}}, (\ell, c)\big)$. As all VPAs that we will consider in the following are 1-SEVPAs, and we will therefore omit an explicit definition of the call transition function.

6.2. A Unified Congruence for Well-Matched VPLs

Alur *et al.* [14] introduce several congruences for well-matched VPLs. The first and most simple one is essentially the syntactic middle congruence on well-matched words. That is, for a well-matched VPL output function $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$, the relation $\approx_\lambda \subseteq \mathrm{WM}(\widehat{\Sigma}) \times \mathrm{WM}(\widehat{\Sigma})$ is defined via

$$w \approx_\lambda w' \Leftrightarrow_{df} \forall u \in \widehat{\Sigma}^*, v \in \widehat{\Sigma}^* : \lambda(u\,w\,v) = \lambda(u\,w'\,v) \quad \forall w, w' \in \mathrm{WM}(\widehat{\Sigma}). \tag{6.1}$$

Note that $u\,w\,v$ is well-matched if and only $u\,w'\,v$ is well-matched. More precisely, these are well matched if and only if $u \in \mathrm{MR}(\widehat{\Sigma})$, $v \in \mathrm{MC}(\widehat{\Sigma})$, and $\beta(u) = -\beta(v)$. Making these constraints explicit is not necessary, as ill-matched words are mapped to 0 under λ, and thus words $u, v \in \widehat{\Sigma}^*$ violating the aforementioned conditions cannot possibly separate words $w, w' \in \mathrm{WM}(\widehat{\Sigma})$ with respect to $\approx_\lambda$.

Holzer and König [87] have shown that for regular languages (which form a subclass of VPLs), the number of equivalence classes of the syntactic middle congruence can be as large as n^n, where n is the number of equivalence classes of the Nerode congruence. Thus, the relation $\approx_\lambda$ is of theoretical interest only: among other things, VPLs can be characterized by having a finite number of equivalence classes with respect to $\approx_\lambda$.

For k-SEVPAs, Alur *et al.* [14] define congruences $\sim_{\lambda,0}$ through $\sim_{\lambda,k}$, each corresponding to one of the $k+1$ modules, that are of much smaller index by imposing restrictions on the role of the prefix u in (6.1). Due to our slightly modified notion of 1-SEVPAs (see above), a single congruence is sufficient for our case. This congruence, again defined on well-matched words, is basically the coarsest refinement of the two equivalence relations $\sim_{\lambda,0}$ and $\sim_{\lambda,1}$ that would result from their definition of a 1-SEVPA.

First, we introduce as an auxiliary definition the concept of *context pairs*.

[4] The term "1-SEVPA" is not used consistently by Alur *et al.* [14]: while a VPA in the aforementioned sense (i.e., the initial location being the target of all call transitions) is referred to as being a 1-SEVPA, the formal definition of k-SEVPA requires that the initial location be part of a module (the "base module", corresponding to an empty stack) that contains no locations which are call targets. We will ignore this technical difference here, and stick with our above definition of 1-SEVPA.

Definition 6.5 (*Context pairs*)

Let $\widehat{\Sigma}$ be a visibly pushdown alphabet. The set of *context pairs* over $\widehat{\Sigma}$, $\mathrm{CP}(\widehat{\Sigma})$, is defined as

$$\mathrm{CP}(\widehat{\Sigma}) =_{df} \left\{ \langle u, v \rangle \in (\mathrm{WM}(\widehat{\Sigma}) \cdot \Sigma_{call})^* \times \mathrm{MC}(\widehat{\Sigma}) \mid \beta(u) = -\beta(v) \right\}.$$

Note that u in the above definition is either the empty word ε, or is from the set $\mathrm{MR}(\widehat{\Sigma}) \cdot \Sigma_{call}$. Furthermore, for each $\langle u, v \rangle \in \mathrm{CP}(\widehat{\Sigma})$, we have $u \cdot v \in \mathrm{WM}(\widehat{\Sigma})$.[5]

We can now proceed to define our unified congruence relation.

Definition 6.6

Let $\lambda : \mathrm{WM}(\widehat{\Sigma}^*) \to \mathbb{B}$ be a well-matched VPL output function. The relation $\simeq_\lambda \subseteq \mathrm{WM}(\widehat{\Sigma}) \times \mathrm{WM}(\widehat{\Sigma})$ is defined via

$$w \simeq_\lambda w' \Leftrightarrow_{df} \forall \langle u, v \rangle \in \mathrm{CP}(\widehat{\Sigma}) : \lambda(u \cdot w \cdot v) = \lambda(u \cdot w' \cdot v)$$

for all $w, w' \in \mathrm{WM}(\widehat{\Sigma})$.

It is easily seen that $\simeq_\lambda$ is a congruence. The following theorem states its significance.

Theorem 6.1

Let $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ be a well-matched output function. λ is a (well-matched) VPL *output function if and only if $\mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda$ is finite.*

Proof: It is obvious that the syntactic middle congruence $\approx_\lambda$ refines $\simeq_\lambda$. If λ is a VPL output function, $\mathrm{WM}(\widehat{\Sigma})/\approx_\lambda$ is finite [14], and thus also $\mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda$.

For the opposite direction, assume that $\mathrm{WM}(\widehat{\Sigma})/\approx_\lambda$ is finite, then define the 1-SEVPA $\mathcal{A} = \langle L_\mathcal{A}, \widehat{\Sigma}, \ell_{0,\mathcal{A}}, \Gamma_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A} \rangle$, where

- $L_\mathcal{A} =_{df} \mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda$,
- $\ell_{0,\mathcal{A}} =_{df} [\varepsilon]_{\simeq_\lambda}$,
- $\Gamma_\mathcal{A} =_{df} L_\mathcal{A} \times \Sigma_{call}$,
- $\delta_\mathcal{A} = \delta_{call,\mathcal{A}} \cup \delta_{ret,\mathcal{A}} \cup \delta_{int,\mathcal{A}}$, where
  - $\delta_{ret,\mathcal{A}}\big([w]_{\simeq_\lambda}, r, ([w']_{\simeq_\lambda}, c)\big) =_{df} [w' c w r]_{\simeq_\lambda} \ \forall w, w' \in \mathrm{WM}(\widehat{\Sigma}), r \in \Sigma_{ret}, c \in \Sigma_{call}$,
  - $\delta_{int,\mathcal{A}}([w]_{\simeq_\lambda}, i) =_{df} [w i]_{\simeq_\lambda} \ \forall w \in \mathrm{WM}(\widehat{\Sigma}), i \in \Sigma_{int}$, and
- $F_\mathcal{A} =_{df} \left\{ [w]_{\simeq_\lambda} \mid \lambda(w) = 1 \right\}$.

Then, $\lambda_\mathcal{A} = \lambda$, and thus λ is a VPL output function.

The previous statement can be proven inductively by showing that, after having read a word $w = w_1 c_1 w_2 c_2 \ldots c_{m-1} w_m \in \mathrm{MR}(\widehat{\Sigma})$, where $c_1, \ldots, c_{m-1} \in \Sigma_{call}$ are the unmatched call symbols in w, and $w_1, \ldots, w_m \in \mathrm{WM}(\widehat{\Sigma})$, the current location is $[w_m]_{\simeq_\lambda}$, and the stack contents are $([w_{m-1}]_{\simeq_\lambda}, c_{m-1}) \ldots ([w_1]_{\simeq_\lambda}, c_1)$. This is trivially guaranteed by construction. Thus, when having read a complete word $w \in \mathrm{WM}(\widehat{\Sigma})$, the stack will be empty, and the location will be $[w]_{\simeq_\lambda}$, which is accepting if and only if $\lambda(w) = 1$. ■

[5] Actually, we even have $\left\{ u \cdot v \mid \langle u, v \rangle \in \mathrm{CP}(\widehat{\Sigma}) \right\} = \mathrm{WM}(\widehat{\Sigma})$.

6.2.1. Finite Characterization

The above theorem is essentially the (well-matched) VPL equivalent for the Myhill-Nerode Theorem (Theorem 3.1). In both cases, the proof is of major importance, as it describes how a respective machine model (DFA or VPA) can be constructed from a congruence relation satisfying certain properties.

The *right-congruence* property of the Nerode congruence allowed an *inductive* characterization of inequivalence of two words $w, w' \in \Sigma^*$, i.e., $w \not\equiv w'$ if and only if $\lambda(w) \neq \lambda(w')$ ("base case") or $wa \not\equiv w'a$ for some $a \in \Sigma$ (inductive step). This observation forms the basis of most minimization algorithms for DFAs, and it also allows us to build the discriminators in a suffix-closed fashion in the TTT algorithm (cf. Section 5.2.2).

The following lemma states a very similar characteristic of our congruence relation $\simeq_\lambda$ defined above.

Lemma 6.1

Let $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ be a well-matched VPL *output function, and let $\simeq_\lambda \subseteq \mathrm{WM}(\widehat{\Sigma}) \times \mathrm{WM}(\widehat{\Sigma})$ be the congruence relation defined in Definition 6.6. Then, the following equivalence holds:*

$$\begin{aligned} w \not\simeq_\lambda w' \Leftrightarrow \big(& \lambda(w) \neq \lambda(w') \\ \vee\ & \exists i \in \Sigma_{int} : w\,i \not\simeq_\lambda w'\,i \\ \vee\ & \exists c \in \Sigma_{call}, r \in \Sigma_{ret}, v \in \mathrm{WM}(\widehat{\Sigma}) : w\,\underbrace{c\,v\,r}\not\simeq_\lambda w'\,\underbrace{c\,v\,r} \\ \vee\ & \exists r \in \Sigma_{ret}, c \in \Sigma_{call}, u \in \mathrm{WM}(\widehat{\Sigma}) : u\,\underbrace{c\,w\,r}\not\simeq_\lambda u\,\underbrace{c\,w'\,r}\big). \end{aligned}$$

Proof: It is obvious that each of these cases implies $w \not\simeq_\lambda w'$, as they form special cases of the (negated) right-hand side of Definition 6.6.

For proving that these cases exhaustively cover all possible ones, let $w, w' \in \mathrm{WM}(\widehat{\Sigma})$ be such that $w \not\simeq_\lambda w'$. According to Definition 6.6, there then exist $\langle u, v \rangle \in \mathrm{CP}(\widehat{\Sigma})$ such that $\lambda(u\,w\,v) \neq \lambda(u\,w'\,v)$ (in particular, we have $\beta(u) = -\beta(v)$). If $v = \varepsilon$, then also $u = \varepsilon$, and thus $\lambda(w) \neq \lambda(w')$. Otherwise, we distinguish the following three cases:

- **Case 1:** v begins with an internal symbol $i \in \Sigma_{int}$. Let v' be such that $v = i\,v'$. We then have $w\,i \not\simeq_\lambda w'\,i$, since $\lambda(u \cdot w\,i \cdot v') \neq \lambda(u \cdot w'\,i \cdot v')$.
- **Case 2:** v begins with a call symbol $c \in \Sigma_{call}$. Let v' be the shortest non-empty, well-matched prefix of v, and define v'' such that $v = v'\,v''$. Note that v' ends with a return symbol $r \in \Sigma_{ret}$, thus we can write v' as $v' = \underbrace{c\,v'''\,r}$. We can conclude that $w\,c\,v'''\,r \not\simeq_\lambda w'\,c\,v'''\,r$, since $\lambda(u \cdot w\,c\,v'''\,r \cdot v'') \neq \lambda(u \cdot w'\,c\,v'''\,r \cdot v'')$.
- **Case 3:** v begins with a return symbol $r \in \Sigma_{ret}$. Define v' such that $v = r\,v'$. This case can only occur if $\beta(u) > 0$, hence $u \neq \varepsilon$. u then ends with a call symbol $c \in \Sigma_{call}$, and we can define u' such that $u = u'\,c$. Let u'' be the longest well-matched suffix of u', and define u''' such that $u' = u'''\,u''$. Observe that $\langle u''', v' \rangle \in \mathrm{CP}(\widehat{\Sigma})$. Therefore, $u''\,c\,w\,r \not\simeq_\lambda u''\,c\,w'\,r$, since $\lambda(u''' \cdot u''\,c\,w\,r \cdot v') \neq \lambda(u''' \cdot u''\,c\,w'\,r \cdot v')$. ∎

Cases 2 and 3 in the above lemma require additional words $u, v \in \mathrm{WM}(\widehat{\Sigma})$ to establish the inequivalence of w and w'. As a learning algorithm in the style of Chapter 3 needs to represent

(an approximation of) $\simeq_\lambda$ in a finite manner, this potentially poses a problem, as arbitrary words in $\mathrm{WM}(\widehat{\Sigma})$ may be required to prove the inequivalence of words $w, w' \in \mathrm{WM}(\widehat{\Sigma})$. The following lemma states that we need not be concerned, as only the equivalence class of the auxiliary words u, v matters.

Lemma 6.2 (*Sufficiency of representatives*)

Let $R \subseteq \mathrm{WM}(\widehat{\Sigma})$ be a set of representatives with respect to $\simeq_\lambda$, i.e., for all $w, w' \in R$, we have $w \neq w' \Rightarrow w \not\simeq_\lambda w'$, and furthermore $\mathrm{WM}(\widehat{\Sigma}) = \bigcup_{w \in R} [w]_{\simeq_\lambda}$. Then, $\mathrm{WM}(\widehat{\Sigma})$ can be substituted with R in Lemma 6.1.

Proof: Define the function $\rho : \mathrm{WM}(\widehat{\Sigma}) \to R$ to map words $w \in \mathrm{WM}(\widehat{\Sigma})$ to their representative elements in R, i.e., ρ is the unique function satisfying $\rho(w) \simeq_\lambda w$ for all $w \in \mathrm{WM}(\widehat{\Sigma})$. We prove the following implications:

$$\forall c \in \Sigma_{call}, r \in \Sigma_{ret}, v \in \mathrm{WM}(\widehat{\Sigma}) : wcvr \not\simeq_\lambda w'cvr \Rightarrow wc\rho(v)r \not\simeq_\lambda w'c\rho(v)r \tag{6.2}$$

and

$$\forall r \in \Sigma_{ret}, c \in \Sigma_{call}, u \in \mathrm{WM}(\widehat{\Sigma}) : ucwr \not\simeq_\lambda ucw'r \Rightarrow \rho(u)cwr \not\simeq_\lambda \rho(u)cw'r \tag{6.3}$$

where $w, w' \in \mathrm{WM}(\widehat{\Sigma})$.

(6.2): Let $c \in \Sigma_{call}, r \in \Sigma_{ret}, v \in \mathrm{WM}(\widehat{\Sigma})$ such that $wcvr \not\simeq_\lambda w'cvr$. Since $v \simeq_\lambda \rho(v)$, we can conclude that both $wc \cdot v \cdot r \simeq_\lambda wc \cdot \rho(v) \cdot r$ and $w'c \cdot v \cdot r \simeq_\lambda wc \cdot \rho(v) \cdot r$, as $\langle wc, r \rangle, \langle w'c, r \rangle \in \mathrm{CP}(\widehat{\Sigma})$. However, with $\simeq_\lambda$ being an equivalence relation, the assumption $wcvr \not\simeq_\lambda w'cvr$ yields $wc\rho(v)r \not\simeq_\lambda w'c\rho(v)r$.

(6.3): Let $c \in \Sigma_{call}, r \in \Sigma_{ret}, u \in \mathrm{WM}(\widehat{\Sigma})$ such that $ucwr \not\simeq_\lambda ucw'r$. Since $u \simeq_\lambda \rho(u)$, we can conclude that both $uc \cdot w \cdot r \simeq_\lambda \rho(u)c \cdot w \cdot r$ and $uc \cdot w' \cdot r \simeq_\lambda \rho(u)c \cdot w' \cdot r$, as $\langle \varepsilon, cwr \rangle, \langle \varepsilon, cw'r \rangle \in \mathrm{CP}(\widehat{\Sigma})$. With $\simeq_\lambda$ being an equivalence relation, the assumption $ucwr \not\simeq_\lambda ucw'r$ yields $\rho(u)cwr \not\simeq_\lambda \rho(u)cw'r$. ∎

6.3. Black-Box Learning of VPLs

In this section, we will pursue the goal of developing an *efficient* active learning algorithm for VPAs. As usual, we assume the existence of a *minimally adequate teacher* (cf. Section 3.2.1) that answers membership and equivalence queries: for the *target function* $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$, which is a well-matched VPL output function, a membership query for a word $w \in \mathrm{WM}(\widehat{\Sigma})$ corresponds to evaluating $\lambda(w)$. An equivalence query, on the other hand, checks if $\lambda_{\mathcal{H}} = \lambda$, where $\mathcal{H}$ is the current hypothesis VPA, and returns a (well-matched) counterexample if the answer is negative.

The learning algorithm we propose always infers 1-SEVPAs. Note that this is merely a choice of how to represent the hypothesis, and does not induce any limitation on the output functions λ (beyond the requirement that it is a well-matched VPL output function). While for certain k-partitions of Σ_{call} the corresponding k-SEVPA may be much smaller than the 1-SEVPA we are

going to infer, the concentration on 1-SEVPA is justified by the fact that they do not require any additional knowledge about the structure of the target systems. Besides, our algorithm can easily be adapted to infer k-SEVPA for a given k-partition of Σ_{call}.

Our aim is to develop a VPA variant of the TTT algorithm. While this may sound like a daunting task, given the complexity of the DFA version of TTT alone, we will see that the previous chapters of this thesis provide us with an extremely powerful "toolbox" that can easily be adapted and enhanced to also work in the VPA case.

The preceding section outlined a clear path towards black-box inference of visibly pushdown languages in the style of Chapter 3: instead of approximating the Nerode congruence $\cong_\lambda$, which for VPLs cannot be assumed to have finite index, we approximate the congruence $\simeq_\lambda$ defined in Definition 6.6, identifying its equivalence classes by means of a finite set of short prefixes $\mathcal{U} \subset \mathrm{WM}(\widehat{\Sigma})$, and using this to construct a hypothesis 1-SEVPA in a way similar to the proof of Theorem 6.1.

Clearly, a more formal transfer of the concepts developed and the phenomena identified in Chapter 3 from regular to visibly pushdown languages is required. We will however dispense with developing a full-fledged framework for active inference of VPLs from scratch, and instead focus on those properties required for our algorithm only. As we are taking the TTT algorithm as a basis, this justifies the following simplifications:

- We assume that the set of representative short prefixes $\mathcal{U}$ is maintained such that its elements are pairwisely inequivalent. That is, we need not concern ourselves with nondeterminism as per Definition 3.9, which is caused by multiple representatives for the same class. Furthermore, this means that a location ℓ has a uniquely defined representative $\lfloor \ell \rfloor \in \mathcal{U}$.
- We assume that $\mathcal{U}$ is maintained in a certain manner (the VPL equivalent of prefix-closedness, as we will define below) that guarantees reachability consistency, i.e., $\mathcal{H}[\lfloor \ell \rfloor] = \ell$ for every hypothesis location (justifying the term *access sequence* for $\lfloor \ell \rfloor$).
- We focus on suffix-based counterexample analysis only (cf. Section 3.3.4), which, thanks to the above two assumptions, can be done using the simplified version described in Remark 3.5.

In the next subsection, we will formalize black-box abstractions for VPLs to the extent needed for developing the algorithm, and under the above simplifying assumptions. This formalization includes a description of counterexample analysis as an instantiation of the abstract framework described in Section 3.3. We will then shift our focus onto the algorithmic realization, and discuss the necessary adaptions for the TTT algorithm, such as data structures and modified discriminator finalization.

6.3.1. Black-box Abstractions for VPLs

We have already sketched above that approximating the Nerode congruence does not make sense in the context of actively learning VPLs, as its index cannot assumed to be finite. Instead, we want to approximate the congruence relation $\simeq_\lambda \subseteq \mathrm{WM}(\widehat{\Sigma}) \times \mathrm{WM}(\widehat{\Sigma})$ as defined in Definition 6.6, and this approximation again is realized by means of a *black-box classifier* (cf. Definition 3.4, p. 28), i.e., a function κ defined on $\mathrm{WM}(\widehat{\Sigma})$ designed in such a way that it is guaranteed that $\simeq_\lambda$ refines $\sim_\kappa$.

Adapting this to the setting of VPLs requires a careful investigation of the differences between the classical Nerode congruence, defined in Definition 3.2 (p. 23), and the congruence $\simeq_\lambda$. Clearly, the role of *suffixes* $v \in \Sigma^*$, which act as witnesses that two words are Nerode-inequivalent, is in our setting assumed by *context pairs* $\langle u, v\rangle \in \mathrm{CP}(\widehat{\Sigma})$, giving rise to the following definition.

Definition 6.7 (VPL *black-box classifier*)

Let $\lambda\colon \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ be a well-matched output function. A VPL *black-box classifier* for λ is a function

$$\kappa\colon \mathrm{WM}(\widehat{\Sigma}) \to \left\{ f\colon \mathrm{CP}(\widehat{\Sigma}) \to \mathbb{B} \mid |\mathrm{dom}\, f| < \infty \right\}.$$

For $w \in \mathrm{WM}(\widehat{\Sigma})$, $Ch_\kappa(w) =_{df} \mathrm{dom}\,\kappa(w)$ denotes the *characterizing set* of w. Furthermore, the set of *separators* of words $w, w' \in \mathrm{WM}(\widehat{\Sigma})$ is defined as

$$Seps_\kappa(w, w') =_{df} \left\{ \langle u, v\rangle \in Ch_\kappa(w) \cap Ch_\kappa(w') \mid \kappa(w)(\langle u, v\rangle) \neq \kappa(w')(\langle u, v\rangle) \right\}.$$

Finally, κ is called *valid* for λ if and only if:

- $\forall w \in \mathrm{WM}(\widehat{\Sigma})\colon \forall \langle u, v\rangle \in Ch_\kappa(w)\colon \kappa(w)(\langle u, v\rangle) = \lambda(u \cdot w \cdot v)$, and
- $\forall w, w' \in \mathrm{WM}(\widehat{\Sigma})\colon w \not\sim_\kappa w' \Rightarrow Seps_\kappa(w, w') \neq \emptyset$.

It is easy to see that the equivalence kernel of a valid black-box classifier κ for an output function λ is refined by the relation $\simeq_\lambda$. Therefore, κ induces an *over-approximation* of $\simeq_\lambda$. We will furthermore implicitly assume that a valid black-box classifier κ satisfies $\forall w \in \mathrm{WM}(\widehat{\Sigma})\colon \langle \varepsilon, \varepsilon\rangle \in Ch_\kappa(w)$, ensuring that $\lambda^{-1}(1)$ is saturated by $\sim_\kappa$.

The step from a black-box classifier to a black-box abstraction (cf. Definition 3.8, p. 29) again involves the introduction of a finite set of representatives for the (rather: some) equivalence classes of $\sim_\kappa$

Definition 6.8 (VPL *black-box abstraction*)

Let $\lambda\colon \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ be a well-matched output function. A VPL *black-box abstraction* for λ is a tuple $\mathcal{R} = \langle \mathcal{U}, \kappa\rangle$, where

- $\mathcal{U} \subset \mathrm{WM}(\widehat{\Sigma})$ is a finite set of *short prefixes* that serve as representatives for the identified equivalence classes, satisfying $\varepsilon \in \mathcal{U}$, and
- κ is a valid VPL black-box classifier for λ.

Before we can construct a VPA from a black-box abstraction $\mathcal{R}$, we need to establish two necessary properties in analogy to Definition 3.9 (p. 29).

Definition 6.9 (*Closedness, determinism*)

Let $\mathcal{R} = \langle \mathcal{U}, \kappa\rangle$ be a VPL black-box abstraction. $\mathcal{R}$ is called ...

1. *closed* if and only if:
   (1.a) $\forall w \in \mathcal{U}, i \in \Sigma_{int}\colon \exists w' \in \mathcal{U}\colon w\, i \sim_\kappa w'$, and
   (1.b) $\forall w, w' \in \mathcal{U}, c \in \Sigma_{call}, r \in \Sigma_{ret}\colon \exists w'' \in \mathcal{U}\colon w\, c\, w'\, r \sim_\kappa w''$.

2. *deterministic* if and only if:
   (2.a) $\forall w, w' \in \mathcal{U}, i \in \Sigma_{int} : w \sim_\kappa w' \Rightarrow w\,i \sim_\kappa w'\,i$,
   (2.b) $\forall w, w', w'' \in \mathcal{U}, c \in \Sigma_{call}, r \in \Sigma_{ret} : w \sim_\kappa w' \Rightarrow w\,c\,w''\,r \sim_\kappa w'\,c\,w''\,r$, and
   (2.c) $\forall w, w', w'' \in \mathcal{U}, c \in \Sigma_{call}, r \in \Sigma_{ret} : w \sim_\kappa w' \Rightarrow w''\,c\,w\,r \sim_\kappa w''\,c\,w'\,r$.

We have already stated above that we will assume in the following that black-box abstractions are always deterministic, simply by maintaining $\mathcal{U}$ as a set of pairwisely inequivalent short prefixes. Still, it is worthwhile to carefully study the definition of this property, as it exhibits the increased complexity compared to the regular case.

If a black-box abstraction $\mathcal{R}$ satisfies these two properties, it is possible to construct a 1-SEVPA from it.

Definition 6.10

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic VPL black-box abstraction. The VPA associated with $\mathcal{R}$ is the 1-SEVPA $\mathcal{H} =_{df} \mathrm{VPA}(\mathcal{R})$, defined via

- $L_\mathcal{H} =_{df} \{[w]_\kappa \mid w \in \mathcal{U}\}$,
- $\ell_{0,\mathcal{H}} =_{df} [\varepsilon]_\kappa$,
- $\Gamma_\mathcal{H} =_{df} L_\mathcal{H} \times \Sigma_{call}$,
- $\delta_\mathcal{H} = \delta_{call,\mathcal{H}} \cup \delta_{ret,\mathcal{H}} \cup \delta_{int,\mathcal{H}}$, where
  - $\delta_{ret,\mathcal{H}}([w]_\kappa, r, ([u]_\kappa, c)) =_{df} [u\,c\,w\,r]_\kappa \ \forall w, u \in \mathcal{U}, c \in \Sigma_{call}, r \in \Sigma_{ret}$,
  - $\delta_{int,\mathcal{H}}([w]_\kappa, i)) =_{df} [w\,i]_\kappa \ \forall w \in \mathcal{U}, i \in \Sigma_{int}$, and
- $F_\mathcal{H} =_{df} \{[w]_\kappa \mid w \in \mathcal{U} \wedge \kappa(w)(\langle \varepsilon, \varepsilon \rangle) = 1\}$.

Note that the above definition mirrors the construction of a 1-SEVPA in the proof of Theorem 6.1, restricted to $\mathcal{U}$.

6.3.2. Consistency Properties

The notion of *reachability consistency* (cf. Definition 3.11, p. 30) was already sketched in the introduction of this section: for every location $\ell \in L_\mathcal{H}$ of $\mathcal{H}$, we denote by $\lfloor \ell \rfloor$ its corresponding representative element in $\mathcal{U}$. Reachability consistency can then be defined as

$$\forall \ell \in L_\mathcal{H} : \mathcal{H}[\lfloor \ell \rfloor] = \ell.$$

In the case of regular languages, reachability consistency could be guaranteed by maintaining $\mathcal{U}$ as a prefix-closed set. It is not entirely obvious how the concept of prefix-closedness can be translated to sets of well-matched words, as not every prefix of a well-matched word is itself well-matched. In addition to the following definition, we will thus explicitly show that it indeed does ensure reachability consistency.

Definition 6.11 (*Well-matched prefix-closedness*)

Let $S \subseteq \mathrm{WM}(\widehat{\Sigma})$ be a set of well-matched words over $\widehat{\Sigma}$. S is called *well-matched prefix-closed*

if and only if the following conditions are satisfied:

(i) if there exist $i \in \Sigma_{int}, w \in \mathrm{WM}(\widehat{\Sigma})$ such that $w\,i \in S$, then also $w \in S$, and

(ii) if there exist $c \in \Sigma_{call}, r \in \Sigma_{ret}, u, w \in \mathrm{WM}(\widehat{\Sigma})$ such that $u\,c\,w\,r \in S$, then also $u, w \in S$.

Lemma 6.3

Let $\mathcal{R} =_{df} \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic VPL *black-box abstraction. If $\mathcal{U}$ is well-matched prefix-closed, then $\mathcal{R}$ is reachability consistent.*

Proof: Assume that $\mathcal{U}$ is well-matched prefix-closed, but $\mathcal{R}$ is reachability inconsistent. Let $w \in \mathcal{U}$ be the shortest word constituting a reachability inconsistency. By definition, the empty word can never constitute a reachability inconsistency. Furthermore, since $w \in \mathrm{WM}(\widehat{\Sigma})$, w can only end with either an internal or a return symbol.

Case 1: w ends with an internal symbol $i \in \Sigma_{int}$, i.e., $w = w'i$ for some $w' \in \mathcal{U}$ (due to well-matched prefix-closedness). Since w was chosen as the shortest element of $\mathcal{U}$ constituting a reachability inconsistency and $|w'| < |w|$, we have $\mathcal{H}[w'] = [w']_\kappa$. However, $\mathcal{H}$ was constructed such that $\delta_{int,\mathcal{H}}([w']_\kappa, i) = [w'i]_\kappa = [w]_\kappa$. Consequently, $\mathcal{H}[w] = [w]_\kappa$, contradicting the assumption that w constituted a reachability inconsistency.

Case 2: w ends with a return symbol $r \in \Sigma_{ret}$, i.e., $w = u\,\underbrace{c\,w'\,r}$ for some $c \in \Sigma_{call}$ and $u, w' \in \mathcal{U}$ due to well-matched prefix-closedness of $\mathcal{U}$. Since u and w' are both shorter than w, they cannot constitute reachability inconsistencies. Thus, $\langle \ell_{0,\mathcal{H}}, \varepsilon \rangle \overset{u}{\Rightarrow} \langle [u]_\kappa, \varepsilon \rangle \xrightarrow{c} \langle \ell_{0,\mathcal{H}}, ([u]_\kappa, c) \rangle \overset{w'}{\Rightarrow} \langle [w']_\kappa, ([u]_\kappa, c) \rangle$. Since by definition $\delta_{ret,\mathcal{H}}\big([w']_\kappa, r, ([u]_\kappa, c)\big) = [u\,c\,w'\,r]_\kappa = [w]_\kappa$, we have $\mathcal{H}[w] = [w]_\kappa$, which again contradicts the assumption that w constituted a reachability inconsistency. ■

Remark 6.2

Just like a prefix-closed set, a finite well-matched prefix-closed set $S \subset \mathrm{WM}(\widehat{\Sigma})$ can be stored in space $\mathcal{O}(|S|)$: every element in S that is not the empty word can be represented by a single internal action and a pointer to another element in S, or a call and a return symbol, combined with two pointers to other elements in S, resulting in constant space per element.

However, there is an important difference: while the length of words in a prefix-closed S set is bounded by $|S|-1$, words in a well-matched prefix closed set can have lengths that are exponential in the size of the set: consider, e.g., the well-matched prefix-closed set $S = \left\{\varepsilon, \underbrace{c\,r}, c\,r\,\underbrace{c\,c\,r\,r}, c\,r\,c\,c\,r\,r\,\underbrace{c\,c\,r\,c\,c\,r\,r\,r}, \ldots\right\} \subset \mathrm{WM}(\widehat{\Sigma})$.

6.3.3. Counterexample Analysis

Motivated by the characteristics of our envisioned algorithm, we only consider the case of suffix-based counterexample analysis in analogy to Section 3.3.4, furthermore simplified by the assumption of unique representatives and guaranteed reachability consistency (cf. Remark 3.5, p. 43).

One of the most important results from Section 3.3.4 (in particular Lemma 3.8) was that suffix-based counterexample analysis is actually analysis of output inconsistencies. While it may be

tempting to translate the definition of output inconsistencies (cf. Definition 3.12, p. 31) directly from the regular case, i.e., by replacing the role of a suffix with a context pair, a slightly modified notion considerably simplifies the presentation: the idea of output inconsistency analysis can be described as pinpointing the transition which, when represented *explicitly* as an element of either $\mathcal{U}\cdot\Sigma_{int}$ or $\mathcal{U}\cdot\Sigma_{call}\cdot\mathcal{U}\cdot\Sigma_{ret}$ (as on the right-hand side of the definition of $\delta_{ret,}$ and $\delta_{int,}$ in Definition 6.10), behaves differently from its successor location, represented as an element of $\mathcal{U}$.

If an output inconsistency was defined as an arbitrary pair $(w,\langle u,v\rangle)\in\mathcal{U}\times\mathrm{CP}(\widehat{\Sigma})$ satisfying $\lambda_{\mathcal{H}}(u\cdot w\cdot v)\neq\lambda(u\cdot w\cdot v)$, a "wrong" transition in either the prefix u or the suffix v could cause the diverging behavior. The following, modified definition of an output inconsistency ensures that we can concentrate on the suffix part, and do not need to worry about the prefix part.

Definition 6.12 ($\mathcal{U}$-*context pair; output inconsistency*)

Let $\mathcal{R}=\langle\mathcal{U},\kappa\rangle$ be a VPL black-box abstraction of some VPL output function $\lambda\colon\mathrm{WM}(\widehat{\Sigma})\to\mathbb{B}$.

1. The set of $\mathcal{U}$*-context pairs*, denoted by $\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$, is defined as the set
$$\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})=_{df}\left\{\langle u,v\rangle\in(\mathcal{U}\cdot\Sigma_{call})^*\times\mathrm{MC}(\widehat{\Sigma})\mid\beta(u)=-\beta(v)\right\}\subset\mathrm{CP}(\widehat{\Sigma}).$$
2. Assume further that $\mathcal{R}$ is closed and deterministic, and let $\mathcal{H}=\mathrm{VPA}(\mathcal{H})$ be its associated VPA. A pair $(w,\langle u,v\rangle)\in\mathcal{U}\times\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitutes an *output inconsistency* if and only if
$$\lambda_{\mathcal{H}}(u\cdot w\cdot v)\neq\lambda(u\cdot w\cdot v).$$

The definition of $\mathcal{U}$-context pairs above precisely accomplishes to eliminate the possibility of incorrect transitions in the prefix part of an output inconsistency, at least if reachability consistency can be assumed.[6]

In the original Definition 3.12, we have furthermore introduced the term "output (in-)consistent" to denote a property of black-box abstractions, referring to whether it is possible to obtain an output inconsistency by combining a short prefix $u\in\mathcal{U}$ with an element of its characterizing set $Ch_{\kappa}(u)$. The above, modified notion of output inconsistencies would only allow a translation of this if $Ch_{\kappa}(u)\subseteq\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ could be ensured. We will see that ensuring this is indeed possible, and furthermore results automatically from a straightforward application of the technique described in the following.

Abstract Counterexample Derivation

One of our stated goals was to leverage the abstract counterexample analysis framework, developed in Section 3.3 of this thesis, for counterexample (or output inconsistency) analysis. As suffix-based analysis is based on the notion of *access sequence transformations* [108], we first need to investigate how this concept translates to the case of VPLs.

[6] Let us briefly sketch what a generalization that does not rely on such assumptions would look like. Assuming that $(w,\langle u,v\rangle)\in\mathcal{U}\times\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitutes an output inconsistency, and $u=u_1c_1u_2c_2\ldots u_mc_m\in(\mathcal{U}\cdot\Sigma_{call})^*$, where $u_i\in\mathcal{U}$, $c_i\in\Sigma_{call}$ for all $1\leq i\leq m$, let $\sigma\in\Gamma_{\mathcal{H}}^*$ denote the stack contents associated with u, i.e., $\sigma=([u_m]_{\kappa},c_m)\cdots([u_1]_{\kappa},c_1)$. The generalized notion of this output inconsistency would then be $\lambda_{\mathcal{H}}^{\langle[w]_{\kappa},\sigma\rangle}(v)\neq\lambda(u\cdot w\cdot v)$, where $\lambda_{\mathcal{H}}^{\langle[w]_{\kappa},\sigma\rangle}$ is the corresponding *state* output function for the state $\langle[w]_{\kappa},\sigma\rangle$ of $\mathcal{H}$.

For a location $\ell \in L_{\mathcal{H}}$, $\lfloor \ell \rfloor \in \mathcal{U}$ denotes its unique representative in $\mathcal{U}$. In Remark 3.5, we have furthermore introduced the notation $\lfloor \cdot \rfloor_{\mathcal{H}}$, defined via $\lfloor u \rfloor_{\mathcal{H}} =_{df} \lfloor \mathcal{H}[u] \rfloor$ (this is the original notion of access sequence transformations).

A *state* in the context of visibly pushdown systems is a more complicated concept, as it comprises not only a (control) location from a finite set, but also stack contents of unbounded length. Let us therefore generalize the concept of an access sequence from locations to states. We start by looking at the stack contents. Let $\sigma = ([u_m]_\kappa, c_m) \cdots ([u_1]_\kappa, c_1) \in (L_{\mathcal{H}} \times \Sigma_{call})^*$, where $u_i \in \mathcal{U}, c_i \in \Sigma_{call}$ for all $1 \leq i \leq m$, be a representation of stack contents. The *access sequence* of σ, $\lfloor \sigma \rfloor$, is defined as the word $\lfloor \sigma \rfloor = u_1 c_1 \dots u_m c_m$. The intuition is that $\lfloor \sigma \rfloor$ is the unique, canonical word in $(\mathcal{U} \cdot \Sigma_{call})^*$ which, when read by $\mathcal{H}$, results in the state $\langle \ell_{0,\mathcal{H}}, \sigma \rangle$. This complements the notion of an access sequence $\lfloor \ell \rfloor$ of a location ℓ, which is the canonical word in $\mathcal{U}$ which, when read by $\mathcal{H}$, results in the state $\langle \ell, \varepsilon \rangle$.[7] Combined, we can define the access sequence of a *state* $\langle \ell, \sigma \rangle \in L_{\mathcal{H}} \times \Gamma$ to be $\lfloor \langle \ell, \sigma \rangle \rfloor = \lfloor \sigma \rfloor \cdot \lfloor \ell \rfloor$. The access sequence *transformation* $\lfloor w \rfloor_{\mathcal{H}}$ of a word $w \in \mathrm{MR}(\widehat{\Sigma})$ is then simply defined as $\lfloor w \rfloor_{\mathcal{H}} =_{df} \lfloor \delta_{\mathcal{H}}(\langle \ell_{0,\mathcal{H}}, \varepsilon \rangle, w) \rfloor$.

Let us discuss some properties of access sequence transformations in the context of VPAs. Due to the 1-SEVPA-property, we have, for arbitrary words $u, u' \in \mathrm{MR}(\widehat{\Sigma})$ and $c \in \Sigma_{call}$, $\lfloor uc \rfloor_{\mathcal{H}} = \lfloor u \rfloor_{\mathcal{H}} c$, and $\lfloor u c u' \rfloor_{\mathcal{H}} = \lfloor u \rfloor_{\mathcal{H}} c \lfloor u' \rfloor_{\mathcal{H}}$. Since reachability consistency ensures that elements of $\mathcal{U}$ are invariant under access sequence transformations, a direct consequence is that all elements of $(\mathcal{U} \cdot \Sigma_{call})^* \cup (\mathcal{U} \cdot \Sigma_{call})^* \cdot \mathcal{U}$ are invariant under access sequence transformations as well.

The following lemma relates counterexamples and output inconsistencies, and states how they can be exploited for refinement. It can thus be regarded as the VPL version of Lemma 3.8 (p. 42).

Lemma 6.4

Let $\mathcal{R} = \langle \mathcal{U}, \kappa \rangle$ be a closed and deterministic VPL *black-box abstraction of some well-matched output function $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ with associated hypothesis $\mathcal{H} = \mathrm{VPA}(\mathcal{R})$.*

(i) If $w \in \mathrm{WM}(\widehat{\Sigma})$ is a counterexample, then $(\varepsilon, \langle \varepsilon, w \rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitutes an output inconsistency.

(ii) If $(w, \langle x, y \rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitutes an output inconsistency, y can be decomposed into $y = \widehat{u}\widehat{a}\widehat{v}$, $\widehat{u}, \widehat{v} \in \widehat{\Sigma}^*, \widehat{a} \in \widehat{\Sigma}$ such that $\lambda(\lfloor x \cdot w \cdot \widehat{u} \rfloor_{\mathcal{H}} \widehat{a} \cdot \widehat{v}) \neq \lambda(\lfloor x \cdot w \cdot \widehat{u}\widehat{a} \rfloor_{\mathcal{H}} \cdot \widehat{v})$.

(iii) Let $(w, \langle x, y \rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitute an output inconsistency, and let $y = \widehat{u}\widehat{a}\widehat{v}$ be a decomposition satisfying the conditions of (ii). Let u be the longest suffix of $w\widehat{u}$ such that $u\widehat{a}$ is well-matched, and let u' be such that $w\widehat{u} = u'u$ (note that $u' \in (\mathrm{WM}(\widehat{\Sigma}) \cdot \Sigma_{call})^*$, thus $\lfloor u' \rfloor_{\mathcal{H}} \in (\mathcal{U} \cdot \Sigma_{call})^*$). Then, $\langle x \cdot \lfloor u' \rfloor_{\mathcal{H}}, \widehat{v} \rangle \in \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ is a $\mathcal{U}$-context pair distinguishing $\lfloor u \rfloor_{\mathcal{H}} \widehat{a}$ and $\lfloor u\widehat{a} \rfloor_{\mathcal{H}}$. Thus, $\lfloor u \rfloor_{\mathcal{H}} \widehat{a} \not\sim_{\kappa'} \lfloor u\widehat{a} \rfloor_{\mathcal{H}}$ for $\kappa' =_{df} \mathrm{split}(\kappa, \mathcal{H}[u\widehat{a}], \langle x \cdot \lfloor u' \rfloor_{\mathcal{H}}, \widehat{v} \rangle)$.

Proof: As usual, we only prove (i) and (iii), and leverage our abstract counterexample analysis framework for proving (ii).

(i) Since $w \in \mathrm{WM}(\widehat{\Sigma})$ is a counterexample, we have $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$. This immediately implies that $(\varepsilon, \langle \varepsilon, w \rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitutes an output inconsistency, as $\lambda_{\mathcal{H}}(\varepsilon \cdot \varepsilon \cdot w) = \lambda_{\mathcal{H}}(w) \neq \lambda(w) = \lambda(\varepsilon \cdot \varepsilon \cdot w)$.

[7] Again, both intuitive descriptions rely on reachability consistency.

(iii) Let $y = \widehat{u}\widehat{a}\widehat{v}$ be a decomposition satisfying the conditions of (ii), let u be the longest suffix of $w\widehat{u}$ such that $u\widehat{a}$ is well-matched, and let u' be such that $w\widehat{u} = u'u$. As noted, $u' \in (\mathrm{WM}(\widehat{\Sigma})\cdot\Sigma_{call})^*$, thus $\lfloor x\,w\,\widehat{u}\rfloor_{\mathcal{H}}\widehat{a} = x\lfloor u'\rfloor_{\mathcal{H}}\cdot\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ and $\lfloor x\,w\,\widehat{u}\,\widehat{a}\rfloor_{\mathcal{H}} = x\lfloor u'\rfloor_{\mathcal{H}}\cdot\lfloor\widehat{u}\,\widehat{a}\rfloor_{\mathcal{H}}$. The conditions of (ii) can thus be written as $\lambda(x\lfloor u'\rfloor_{\mathcal{H}}\cdot\lfloor u\rfloor_{\mathcal{H}}\widehat{a}\cdot\widehat{v}) \neq \lambda(x\lfloor u'\rfloor_{\mathcal{H}}\cdot\lfloor\widehat{u}\,\widehat{a}\rfloor_{\mathcal{H}}\cdot\widehat{v})$, yielding that $\langle x\cdot\lfloor u'\rfloor_{\mathcal{H}}, \widehat{v}\rangle$ is a context pair separating $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ and $\lfloor u\widehat{a}\rfloor_{\mathcal{H}}$. ■

It should be noted that Lemma 6.4 (iii) again provides instructions on how to exploit the result of an output inconsistency analysis to refine the VPL black-box abstraction. In particular, the context pair that is used for splitting a class in κ (resulting in the refined classifier κ') is always in $\mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$, thus maintaining $Ch_{\kappa'}(u) \subseteq \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ for all $u \in \mathrm{WM}(\widehat{\Sigma})$.

Another important observation is that adding $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ to $\mathcal{U}$ preserves well-matched prefix-closedness of $\mathcal{U}$: if $\widehat{a} \in \Sigma_{int}$, then $u \in \mathrm{WM}(\widehat{\Sigma})$ and thus $\lfloor u\rfloor_{\mathcal{H}} \in \mathcal{U}$. In this case, $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ is the access sequence of the internal $\widehat{a}$-transition of $\mathcal{H}[u]$. Otherwise, $\widehat{a}$ must be an element of Σ_{ret}, and $u\widehat{a}$ being well-matched implies $\lfloor u\rfloor_{\mathcal{H}} \in \mathcal{U}\cdot\Sigma_{call}\cdot\mathcal{U}$. Let $\lfloor u\rfloor_{\mathcal{H}} = \widetilde{u}\,c\,\widetilde{u}'$, then $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ is the $\widehat{a}$-return transition of $\mathcal{H}[\widetilde{u}']$ for the stack symbol $(\mathcal{H}[\widetilde{u}], c)$. Therefore, adding $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ to $\mathcal{U}$ can in both cases be realized by converting a non-tree transition of $\mathcal{H}$ into a tree transition. Furthermore, the construction of $\mathcal{H}$ implies that $\lfloor u\rfloor_{\mathcal{H}}\widehat{a}$ and $\lfloor u\widehat{a}\rfloor_{\mathcal{H}}$ where equivalent wrt. $\sim_{\kappa}$, thus the refined classifier $\kappa' =_{df} \mathrm{split}\big(\kappa, \mathcal{H}[u\widehat{a}], \langle x\cdot\lfloor u'\rfloor_{\mathcal{H}}, \widehat{v}\rangle\big)$ is a strict refinement of κ.

Let us now take a look at how our abstract counterexample framework can be leveraged to obtain a decomposition with the properties stated in Lemma 6.4 (ii).

Definition 6.13 (*Derived abstract counterexample*)

Let $\mathcal{R} = \langle\mathcal{U},\kappa\rangle$ be a closed and deterministic VPL black-box abstraction of some well-matched output function $\lambda\colon \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ with associated hypothesis $\mathcal{H} = \mathrm{VPA}(\mathcal{R})$. For a pair $(w, \langle x, y\rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$, the *derived abstract counterexample* is the abstract counterexample $\alpha = \langle\mathbb{B}, =, |y|, \eta\rangle$, where the effect mapping is defined as

$$\eta\colon \{0,\ldots,|y|\} \to \mathbb{B}, \quad \eta(i) =_{df} \lambda(\lfloor x\cdot w\cdot y_{1..i}\rfloor_{\mathcal{H}}\cdot y_{i+1..|y|}).$$

Lemma 6.5

Let $\mathcal{R} = \langle\mathcal{U},\kappa\rangle$ be a closed and deterministic VPL *black-box abstraction of some well-matched output function $\lambda\colon \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$, let $\mathcal{H} = \mathrm{DFA}(\mathcal{R})$ be its associated* VPA*, and let $(w, \langle x, y\rangle) \in \mathcal{U} \times \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$ constitute an output inconsistency, i.e., $\lambda_{\mathcal{H}}(x\cdot w\cdot y) \neq \lambda(x\cdot w\cdot y)$. Then, the abstract counterexample α derived according to the above Definition 6.13 is valid, and if i is a breakpoint in α, $\widehat{u} = y_{1..i}$, $\widehat{a} = y_{i+1}$, $\widehat{v} = y_{i+2..|y|}$ is a decomposition of y satisfying the conditions of Lemma 6.4 (ii).*

Proof: We start by showing that the derived abstract counterexample α is valid. First, observe that $\lfloor x\cdot w\rfloor_{\mathcal{H}} = x\cdot w$ as remarked above (since $x \in (\mathcal{U}\cdot\Sigma_{call})^*, w \in \mathcal{U}$). Thus, $\eta(0) = \lambda(x\cdot w\cdot y) \neq \lambda_{\mathcal{H}}(x\cdot w\cdot y)$, due to $(w, \langle x, y\rangle)$ constituting an output inconsistency. On the other hand, $\eta(|y|) = \lambda(\lfloor x\cdot w\cdot y\rfloor_{\mathcal{H}}) = \lambda_{\mathcal{H}}(x\cdot w\cdot y)$, since the location represented by $\lfloor x\cdot w\cdot y\rfloor_{\mathcal{H}\,df} = u \in \mathcal{U}$ is accepting if and only if $\kappa(u)(\langle\varepsilon,\varepsilon\rangle) = \lambda(u) = 1$. We therefore have established that $\eta(0) \neq \eta(|y|)$

The fact that the decomposition corresponding to a breakpoint satisfies the conditions of Lemma 6.4 (ii) follows directly from the definition of η and the breakpoint condition. ■

Remark 6.3

The observed property that, for arbitrary $u \in \mathrm{MR}(\widehat{\Sigma})$ and $c \in \Sigma_{call}$, we have $\lfloor u\,c \rfloor_{\mathcal{H}} = \lfloor u \rfloor_{\mathcal{H}} c$, implies that positions i corresponding to call symbols in y (i.e., satisfying $y_{i+1} \in \Sigma_{call}$) can never be breakpoints. This can be exploited to derive a marginally smaller abstract counterexample in which these positions are eliminated.

6.4. A VPDA Version of TTT

After describing how certain key concepts—black-box abstractions, hypothesis construction, and counterexample analysis—of the framework developed in Chapter 3 can be transferred to the setting of VPLs, we can now describe how the TTT algorithm presented in Chapter 5 can be adapted to learn visibly pushdown automata, or, more precisely, 1-SEVPAs. Since TTT has already been described in great technical detail, the description in this section will remain incremental, i.e., only elaborating on the differences and necessary adaptions.

One of the motivations for developing the TTT algorithm was to reduce the overall length of queries, especially in the presence of non-minimal counterexamples. This is of even greater importance in the context of VPAs: as Kumar *et al.* [119] observe, even a *cooperative teacher* [175] might be forced to provide counterexamples of length exponential in the size of the target 1-SEVPA, which also means that if techniques like random sampling are used for approximating equivalence queries, sampled words of considerable length need to be generated in order to achieve a reasonable chance of finding counterexamples. This, in turn, results in an increased probability of generating counterexamples that are much longer than minimal ones, which calls for attempts to shorten the length of queries.

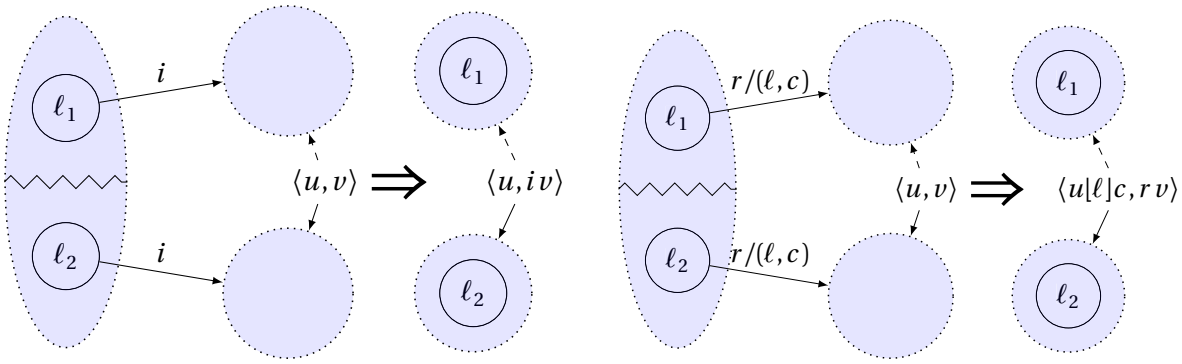
However, the above observation also means that posing queries of exponential length might be inevitable. Thus, all sophisticated finalization techniques cannot help the symbol complexity becoming exponential in the worst case. We will detail on the guarantees that *can* be made in Section 6.4.5.

6.4.1. Data Structures

In the previous Section 6.3, we have observed that when learning VPLs, *context pairs* instead of suffixes assume the role of discriminators. This provides a clear guideline on how the discrimination tree data structure needs to be changed: inner nodes are no longer labeled with a single suffix v, but with a context pair $\langle u, v \rangle \in \mathrm{CP}_{\mathcal{U}}(\widehat{\Sigma})$. When sifting a word w (such as a transition access sequence) into the tree, at each inner node labeled with $\langle u, v \rangle$, the outcome of the membership query $\lambda(u \cdot w \cdot v)$ determines the successor node. The general notions of *soft* and *hard* sifting, as well as of *temporary discriminators* (cf. Section 5.2.1) remain unaffected.

What about the *spanning-tree hypothesis* (cf. Section 4.2.2)? As mentioned before, the hypothesis will always be maintained as a 1-SEVPA, which in particular means that we can omit explicitly specifying any call transitions. The remaining internal and return transitions can again be either tree or non-tree transitions, with the tree transitions forming a *spanning-tree*, rooted at the initial location.

Access sequences are assigned to locations and transitions as follows. The initial location has the access sequence ε. Every outgoing i-transition ($i \in \Sigma_{int}$) of a location ℓ (the access sequence



(a) Finalization rule for internal transitions

(b) Finalization rule for returns

Figure 6.2.: Abstract visualization of discriminator finalization rules for internal and return actions

of which is denoted by $\lfloor\ell\rfloor$) is assigned the access sequence $\lfloor\ell\rfloor i$. An outgoing return-transition of ℓ labeled $r/(\ell', c)$, where $r \in \Sigma_{ret}, c \in \Sigma_{call}$ and $\ell' \in L_{\mathcal{H}}$, is assigned the access sequence $\lfloor\ell'\rfloor c\lfloor\ell\rfloor r$. The access sequences of locations other than the initial one, finally, are the access sequences of their unique incoming tree transition.

6.4.2. Discriminator Finalization

The impressive practical efficiency of TTT is mostly due to its *discriminator finalization* step, i.e., replacing the "temporary" discriminators that are extracted directly from the counterexample with discriminators that are derived from the known transition structure of the hypothesis. In principle, the discriminator finalization step can be regarded as a refinement step during DFA minimization, where the current partition is given by the set of blocks in the discrimination tree.

We have already remarked in Section 6.2.1 that the basis for this is the "inductive" characterization of inequivalence wrt. the Nerode congruence, i.e., for two words w, w' which are Nerode-inequivalent but still satisfy $\lambda(w) = \lambda(w')$, a separating word for them can be obtained from a separator of one of their a-successors ($a \in \Sigma$). This is directly reflected in the visualization of the finalization rule shown in Figure 5.6.

For visibly pushdown languages and the congruence $\simeq_\lambda$ as defined in Definition 6.6, a very similar yet slightly more complex approach is possible. Instead of one finalization rule as in the case of regular languages,[8] when learning VPLs there are three different rules that may apply, each one corresponding to one of the non-trivial disjuncts on the right-hand side of the equivalence in Lemma 6.1. Two of these rules, namely the one for internal and return transitions, are visualized in Figure 6.2. The rule for internal actions (Figure 6.2a) is almost a straightforward adaption of the finalization rule for the regular case, with the exception that the prefix part of the context pair (i.e., u) separating the successors needs to be present in the new separator for ℓ_1 and ℓ_2 as well. The rule for return transitions (Figure 6.2b) requires modifying this prefix, to ensure that the topmost element on the stack allows triggering the considered r-transition.

The rule for *calls*, depicted in Figure 6.3, is somewhat more complicated. This is due to the

[8] Or two, if the extension to Mealy machines as described in Section 5.4 is considered.

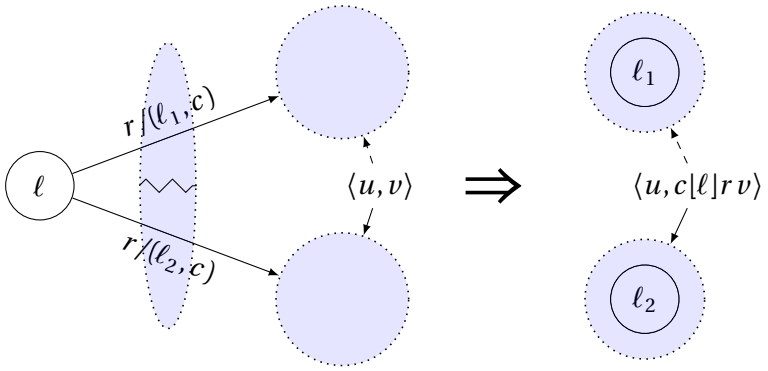


Figure 6.3.: Abstract visualization of discriminator finalization rule for calls

fact that there are no (meaningful) call transitions in 1-SEVPA. For this reason, the locations ℓ_1 and ℓ_2 that should be separated by a final discriminator are not the source locations of these transitions, but instead are part of the return transition (in the form of the stack symbol) of some other location.

6.4.3. Progress and Subsequent Splits

An important insight during the development of TTT was that there may be situations in which none of the finalization rules are applicable, which however implies that an output inconsistency *must* be present. Reasoning about output inconsistencies was however only possible because the inapplicability of any finalization rule (as formally characterized by condition (5.1) on p. 94) guaranteed that the possibly *non-deterministic* hypothesis behaves *deterministically* up to the granularity of the block structure.

To see that the same is true in the case of visibly pushdown languages, let us first formally characterize the situation that none of the rules from Figures 6.2 and 6.3 are applicable. As usual, $\pi(\mathcal{T})$ denotes the block partition induced by the block subtrees of the discrimination tree $\mathcal{T}$.

$$\begin{aligned}
&\forall B \in \pi(\mathcal{T}): \\
&\quad \Big(\quad \forall i \in \Sigma_{int} : \exists B' \in \pi(\mathcal{T}) : \delta_{int,\mathcal{H}}(B,i) \subseteq B' \\
&\wedge \quad\quad \forall \ell \in L_{\mathcal{H}}, r \in \Sigma_{ret}, c \in \Sigma_{call} : \exists B' \in \pi(\mathcal{T}) : \delta_{ret,\mathcal{H}}\big(B,(\ell,c)\big) \subseteq B' \\
&\wedge \quad\quad \forall \ell \in L_{\mathcal{H}}, c \in \Sigma_{call}, r \in \Sigma_{ret} : \exists B' \in \pi(\mathcal{T}) : \delta_{ret,\mathcal{H}}\big(\ell,(B,c)\big) \subseteq B' \Big),
\end{aligned} \tag{6.4}$$

where the (non-deterministic) transition functions are lifted to sets of locations in the usual fashion, i.e.:

$$\begin{aligned}
\delta_{int,\mathcal{H}}(B,i) &=_{df} \bigcup_{\ell' \in B} \delta_{int,\mathcal{H}}(\ell',i) && \forall B \in \pi(\mathcal{T}), i \in \Sigma_{int}, \\
\delta_{ret,\mathcal{H}}\big(B,r,(\ell,c)\big) &=_{df} \bigcup_{\ell' \in B} \delta_{ret,\mathcal{H}}\big(\ell',r,(\ell,c)\big) && \forall B \in \pi(\mathcal{T}), r \in \Sigma_{ret}, c \in \Sigma_{call}, \ell \in L_{\mathcal{H}}, \\
\delta_{ret,\mathcal{H}}\big(\ell,r,(B,c)\big) &=_{df} \bigcup_{\ell' \in B} \delta_{ret,\mathcal{H}}\big(\ell,r,(\ell',B)\big) && \forall B \in \pi(\mathcal{T}), c \in \Sigma_{call}, r \in \Sigma_{ret}, \ell \in L_{\mathcal{H}}.
\end{aligned}$$

Again, this allows us to define a (deterministic!) output function $\lambda_{\mathcal{H}} : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$, and thus enables us to reason about output inconsistencies. These certainly must exist, as for two locations $\ell_1 \neq \ell_2$ within the same block, there exists a separator $\langle u, v \rangle$, which is the label of their

(temporary) lowest common ancestor, proving $\lambda(u\cdot\lfloor\ell_1\rfloor\cdot v)\neq\lambda(u\cdot\lfloor\ell_2\rfloor\cdot v)$. Due to the above condition, however, we know that $\lambda_{\mathcal{H}}(u\cdot\lfloor\ell_1\rfloor\cdot v)=\lambda_{\mathcal{H}}(u\cdot\lfloor\ell_2\rfloor\cdot v)$, thus either $(\ell_1,\langle u,v\rangle)$ or $(\ell_2,\langle u,v\rangle)$ must constitute an output inconsistency.

This output inconsistency can then be exploited for further refining the abstraction by splitting a leaf in the tree, as described in Section 6.3.3. However, since the abstraction induced by the discrimination tree can never refine $\simeq_\lambda$, (6.4) must eventually be violated—assuming that λ is a VPL output function, i.e., $\mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda$ is finite—, enabling a finalization step.

6.4.4. An Example Run

We will omit a complete pseudocode listing of the algorithm here, as the description should allow to easily infer the necessary modifications for the TTT algorithm and its data structures. We will instead demonstrate a run of the algorithm on a small example, namely the VPL $\mathcal{L}=_{df}\{c^m i r^m \mid m\in\mathbb{N}\}$ over the visibly pushdown alphabet $\widehat{\Sigma}=\langle\Sigma_{call},\Sigma_{ret},\Sigma_{int}\rangle$, where $\Sigma_{call}=_{df}\{c\}$, $\Sigma_{ret}=_{df}\{r\}$, and $\Sigma_{int}=_{df}\{i\}$.

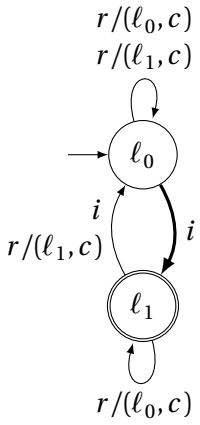
The algorithm starts with the initial hypothesis shown in Figure 6.4a, where the accepting location ℓ_1 with access sequence i is discovered *en passant* during initialization. The corresponding discrimination tree is shown in Figure 6.4b. This hypothesis erroneously accepts the word $ccicrirr\notin\mathcal{L}$. Analysis of the counterexample shows that $\lambda_{\mathcal{L}}(\lfloor ccic\rfloor_{\mathcal{H}}r\cdot irr)=\lambda_{\mathcal{L}}(ccicr\cdot irr)\neq\lambda_{\mathcal{L}}(\lfloor ccicr\rfloor_{\mathcal{H}}\cdot irr)=\lambda_{\mathcal{L}}(cc\cdot irr)$. Following the description given in Section 6.3.3, the leaf corresponding to ℓ_0 is split, using the context pair $\langle cc,irr\rangle$ as the temporary discriminator, and a new location ℓ_2 with access sequence $icir$ is introduced. This results in the hypothesis (omitting most non-deterministic transitions) and discrimination tree as shown in Figure 6.4c. The *block targets* of selected transitions are visualized using dotted lines.

This situation now allows two different ways of replacing the temporary discriminator: the i-transitions of ℓ_0 and ℓ_2 point into separate blocks (corresponding to the rule shown in Figure 6.2a), and the r-transition of ℓ_1 points into different blocks depending on whether the stack symbol is (ℓ_0,c) or (ℓ_2,c) (corresponding to the rule from Figure 6.3). Exploiting the former results in the discrimination tree shown in Figure 6.5a, while the latter results in the one shown in Figure 6.5b. Regardless of which way is chosen, the resulting (final) hypothesis is the same, namely the one shown in Figure 6.5c.

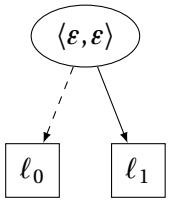
6.4.5. Complexity

Let us now take a closer look at the complexity of TTT-VPA. Assume that n is the number of locations in the canonical 1-SEVPA for λ, i.e., $n=_{df}|\mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda|$. Obviously, the worst-case depth of a corresponding discrimination tree is $n-1$. The number of transitions (only considering relevant ones, i.e., ignoring call transitions) is $n\big(|\Sigma_{int}|+n\cdot|\Sigma_{call}|\cdot|\Sigma_{ret}|\big)$: for each return symbol in Σ_{ret}, there are $n\cdot|\Sigma_{call}|$ possible stack symbols to consider. Since furthermore every counterexample results in an increase in the number of equivalence classes, it is clear that $n-1$ equivalence queries are sufficient.

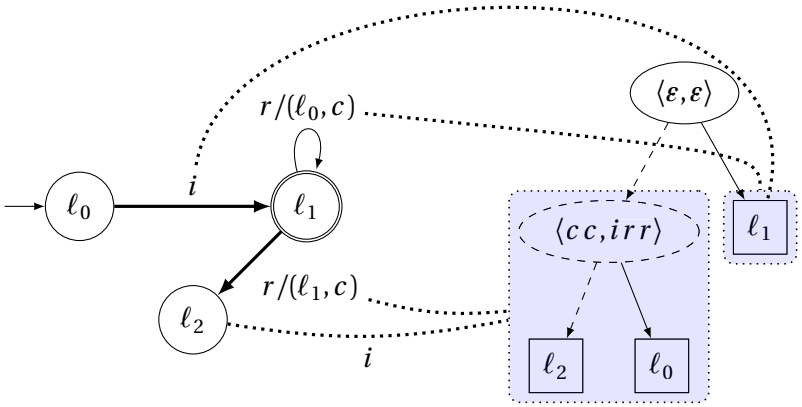
Query complexity. Obviously, sifting transitions down the discrimination tree dominates the query complexity ($\mathcal{O}(n)$ queries per transition). Hard sifts do not increase the number of queries asymptotically, as we have already pointed out in Section 5.3.1. Counterexample analysis using binary search (cf. Proposition 3.3) requires $\mathcal{O}(\log m)$ queries per counterexample, thus



(a) Initial hypothesis

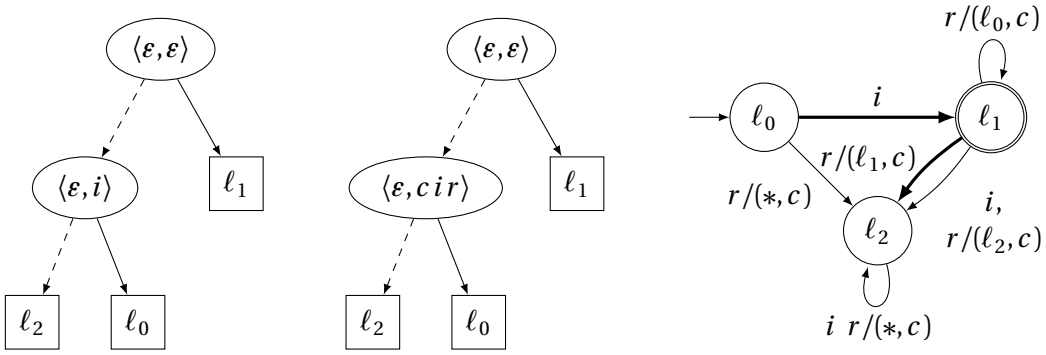


(b) Initial discrimination tree



(c) Non-deterministic hypothesis and corresponding discrimination tree after split

Figure 6.4.: TTT-VPA data structures during a run on $\mathcal{L} = \{c^m i r^m \mid m \in \mathbb{N}\}$ until first split



(a) Discrimination tree after finalization based on i-transitions

(b) Discrimination tree after finalization based on r-transition of ℓ_1

(c) Final hypothesis

Figure 6.5.: Possible final discrimination trees and final hypothesis during a run of TTT-VPA on $L = \{c^m i r^m \mid m \in \mathbb{N}\}$

$\mathcal{O}(n \log m)$ queries in total, resulting in an overall query complexity of $\mathcal{O}(n^2 \cdot |\Sigma_{int}| + n^3 \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}| + n \log m)$.

Proposition 6.1

TTT-VPA *correctly infers a* 1-SEVPA *model of some well-matched* VPL *target output function* $\lambda : \mathrm{WM}(\widehat{\Sigma}) \to \mathbb{B}$ *using at most* $n-1$ *equivalence queries and* $\mathcal{O}(n^2 \cdot |\Sigma_{int}| + n^3 \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}| + n \log m)$ *membership queries, where* $n =_{df} |\mathrm{WM}(\widehat{\Sigma})/\simeq_\lambda|$ *is the size of the canonical* 1-SEVPA *for* λ.

Symbol complexity. We have already remarked in the introduction of this section that queries of exponential length may be inevitable. This is due to the fact that a well-matched prefix-closed set (such as $\mathcal{U}$) of size n may contain words of exponential length (up to $2^n - 2$). However, the finalization steps (Figures 6.2 and 6.3) ensure that the combined length of every context pair in the final discrimination tree is in $\mathcal{O}(n\ell)$, where ℓ is the length of the longest element in $\mathcal{U}$. For counterexample analysis, finally, the worst-case estimate is that a prefix of m reaches a state with a stack of size $\mathcal{O}(m)$, the access sequence of which thus may have a length in $\mathcal{O}(m\ell)$. As a consequence, temporary discriminators of length $\mathcal{O}(m\ell)$ might be extracted from counterexamples.

If no hard sifts are ever necessary during learning, the symbol complexity is $\mathcal{O}(n^3\ell \cdot |\Sigma_{int}| + n^4\ell \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}| + n\ell m \log m)$. Under the worst-case assumption that hard sifts are necessary for every transition and temporary discriminator, this increases to $\mathcal{O}\big(n^2\ell(n+m) \cdot |\Sigma_{int}| + n^3\ell(n+m) \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}| + n\ell m \log m\big)$.

Space complexity. The space complexity of TTT-VPA is again dominated by the size of the hypothesis, i.e., $\Theta(n \cdot |\Sigma_{int}| + n^2 \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}|)$. The set $\mathcal{U}$ of location access sequences is stored implicitly as part of the spanning-tree hypothesis, and all discriminators combined require $\Theta(n)$ space: in each of the finalization rules shown in Figures 6.2 and 6.3, the new discriminator is derived from a previously existing discriminator, combined with either an internal action $i \in \Sigma_{int}$ (Figure 6.2a), or a call symbol, a return symbol, and an element of $\mathcal{U}$ (Figures 6.2b and 6.3). Thus,

a constant number of symbols and pointers is required for each new discriminator, and since there can never be more than $n-1$ discriminators, all (finalized) discriminators can be stored using $\Theta(n)$ space.

6.5. Preliminary Evaluation

Visibly pushdown automata are a relatively recent formalism, and unlike in the case of finite-state machines, there is no collection of publicly available models that would make for interesting benchmarks, at least not to the knowledge of the author. Besides, there do not exist any other learning algorithms that could be used for comparison: attempting to implement the algorithm described by Kumar *et al.* [119] for 1-SEVPAs resulted in errors, and a closer inspection of the algorithm revealed that the description is probably incomplete.[9]

6.5.1. Experimental Setup

The approach we have thus taken is similar to Section 5.5.3: we randomly generated 1-SEVPAs over certain alphabets, minimized them, and compared the performance of learning algorithms for growing counterexample sizes. To illustrate the impact of the discriminator finalization, we compared TTT-VPA against an Observation Pack version for VPAs (i.e., omitting the discriminator finalization steps in the algorithm described in the previous section). Exponential search was used as a search strategy for finding breakpoints when analyzing (abstract) counterexamples. We furthermore used a cache in all of the experiments such that only unique queries (and the symbols occurring in these) were counted, even though no significant amount of redundant queries could be observed.

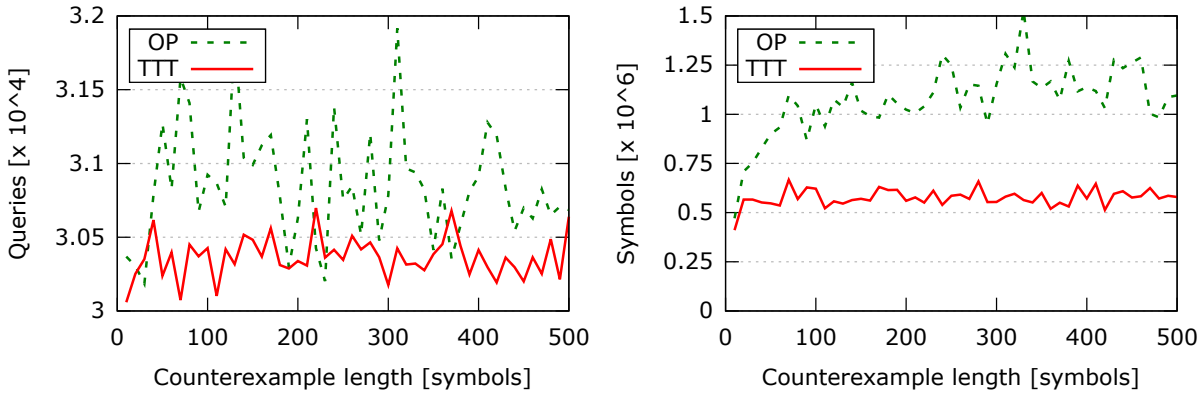
When presented with minimal counterexamples, both algorithms showed a very similar performance, which reflects our findings from the case of DFAs (cf. Section 5.5.2). We will thus only consider the case of non-minimal counterexamples.

6.5.2. Counterexamples of Growing Length

For the first series experiments, we randomly generated a minimal 1-SEVPA with 50 locations over the alphabet $\Sigma_{call} = \{c_1, c_2\}$, $\Sigma_{ret} = \{r_1\}$ and $\Sigma_{int} = \{a, b\}$, and randomly generated (well-matched) counterexamples of lengths between 10 and 500, in increments of 10. We measured the queries and symbol complexities, averaged over 5 runs for each counterexample length.

The results can be seen in Figure 6.6. The VPA version of Observation Pack and TTT-VPA make roughly the same number of membership queries, with Observation Pack showing a considerably higher variance. As in the regular case, the number of queries seems to be virtually unaffected by the length of the counterexample, which probably is due to the nature of randomly generated automata (cf. Section 5.5.3). Looking at the number of symbols, TTT-VPA requires roughly half as many symbols as Observation Pack for counterexamples exceeding a length of 100. Again, Observation Pack shows a significantly greater variance. An interesting aspect is

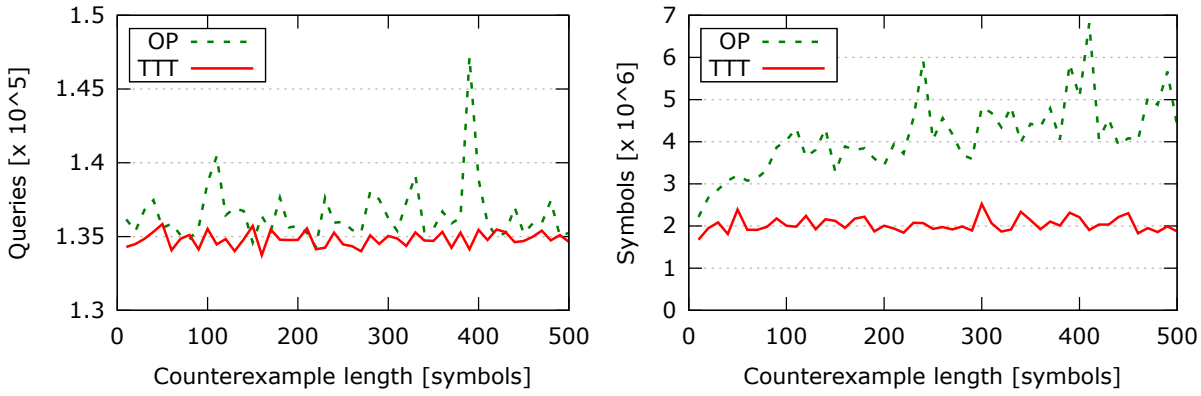
[9] In particular, the pseudocode listing of the discrimination tree refinement (to be found on p. 23 of the accompanying technical report [120]) only constructs discriminators by prepending a symbol to the *suffix* of a context pair, except for in a boundary condition occurring only once per module (with the root of the initial module always being labeled with $\langle \varepsilon, \varepsilon \rangle$). It is clear that this cannot be sufficient, as it—in case of a single module—means that only the syntactical right-congruence (i.e., Nerode congruence) is approximated.



(a) Unique queries

(b) Symbols in unique queries

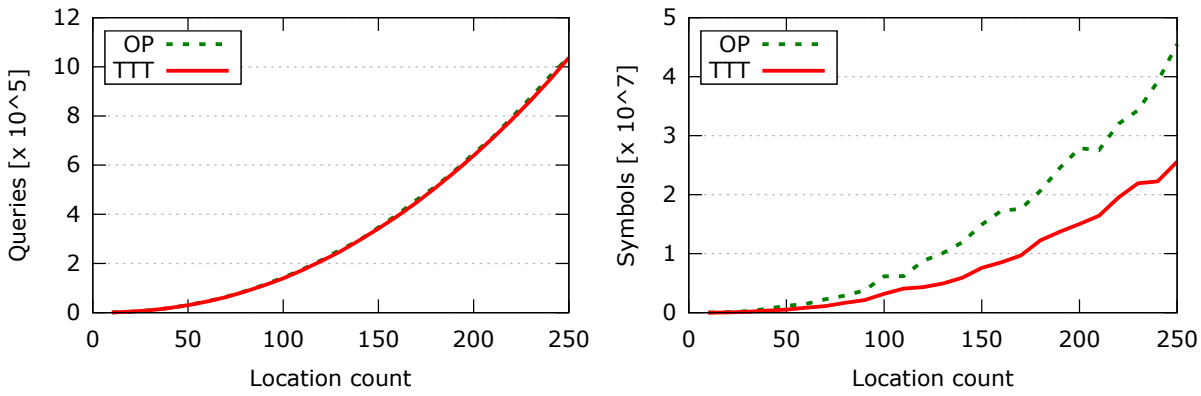
Figure 6.6.: Performance of 1-SEVPA learning algorithms for randomly generated 1-SEVPA with $n = 50$, $|\Sigma_{call}| = |\Sigma_{int}| = 2$, $|\Sigma_{ret}| = 1$



(a) Unique queries

(b) Symbols in unique queries

Figure 6.7.: Performance of 1-SEVPA learning algorithms for randomly generated 1-SEVPA with $n = 50$, $|\Sigma_{call}| = |\Sigma_{ret}| = 3$, $|\Sigma_{int}| = 2$



(a) Unique queries

(b) Symbols in unique queries

Figure 6.8.: Performance of 1-SEVPA learning algorithms as a function of n

that for both algorithms, increasing the length of counterexamples beyond 150 seems to only marginally affect the number of symbols.

We then looked at how increasing the alphabet size affects the performance. We introduced one additional call symbol and two additional return symbols, resulting the alphabet $\Sigma_{call} = \{c_1, c_2, c_3\}$, $\Sigma_{ret} = \{r_1, r_2, r_3\}$ and $\Sigma_{int} = \{a, b\}$ (note that this corresponds to a 4.5-fold increase in the number of return transitions). The results are shown in Figure 6.7. Apart from the total numbers, these results do not differ significantly from the above: the number of membership queries is within a very close range, but the queries posed by Observation Pack contain roughly twice as many symbols as the queries posed by TTT-VPA.

6.5.3. Automata of Growing Size

For the last series of experiments, we randomly generated 1-SEVPAs over the smaller alphabet (i.e., $\Sigma_{call} = \{c_1, c_2\}$, $\Sigma_{ret} = \{r_1\}$, $\Sigma_{int} = \{a, b\}$) with sizes (i.e., number of locations) between 10 and 250, in increments of 10. Counterexamples of a fixed length of $m = 200$ were provided to the learning algorithms. Unique queries and symbols were then measured, again averaging over five runs.

The results, shown in Figure 6.8, are in line with the expectations set by the previous experiments. The number of unique queries is almost the same for both algorithms. Compared to the similar setting in the regular case (as shown in Figure 5.16a), one notices immediately that the growth is no longer near-linear, but instead quadratic. This is due to the fact that the number of transitions grows quadratically with the number of locations, as there are $n \cdot |\Sigma_{call}| \cdot |\Sigma_{ret}|$ return transition per location (cf. also Section 6.4.5).

Finally, the plot depicting the number of symbols in all unique queries (Figure 6.8b) shows the familiar pattern that discriminator finalization, as implemented in TTT-VPA, reduces this number by roughly 50%.

6.5.4. Interpretation of the Results

The presented results underline two aspects: first, applying what can be called the "TTT principle", i.e., cleaning up the internal data structures to represent information about the hypothesis in a minimal form, also pays off in the context of visibly pushdown systems. This is only marginally visible when considering the number of membership queries (which most likely is due to the characteristics of randomly generated systems, cf. Section 5.5.4), but the difference grows significantly when considering the number of symbols in these queries.

Second, discriminator finalization seems to have a very *stabilizing* effect. in the sense that it considerably reduces the variance of the number of both symbols and queries. Clearly, this is due to the fact that TTT-VPA effectively reduces counterexamples, which were the main source of randomness in the above experiments, to a minimal form.

The evaluation can however only be regarded as very preliminary. The problem that randomly generated automata might not be that representative for realistic systems, as already stated in Section 5.5.4, is even more grave in the case of visibly pushdown automata. In general, we can expected TTT-VPA to perform even stronger, as the experiments presented in Section 5.5 suggest that the TTT principle often leads to a reduction in the number of queries, an effect that cannot be observed in the case of randomly generated systems due to the fact that any randomly

sampled discriminator results in an expected balanced partitioning of location sets. In fact, the curve from Figure 6.8a is close to the optimal query complexity, i.e., assuming a discrimination tree depth of $\log n$ (cf. also Section 6.4.5).

Furthermore, factors such as the alphabet size, or the number of call, return, and internal symbols (and their ratios) all can have a significant impact on the performance of the learning algorithm. Conducting larger, more realistic case studies is thus inevitable to make a more realistic assessment regarding not only of the performance of specific learning algorithms, but concerning the applicability and feasibility of learning visibly pushdown systems in general. However, the results in the previous section do suggest that for these case studies, TTT-VPA will probably be the way to go.

6.6. Envisioned Applications

Let us conclude our chapter on learning visibly pushdown systems with sketching two possible applications.

XML document processing. Visibly pushdown languages (sometimes also called languages over *nested words*) have often been proposed [11, 12] for modeling the contents XML documents (more precisely: the set of all XML documents that are valid wrt. some specification). Here, the call symbols correspond to *opening tags*, whereas return symbols correspond to *closing tags*.[10] While it is common to model XML document processing using tree automata (cf. the survey by Schwentick [159]), this requires the document to be present in the form of a tree (such as a DOM tree). A visibly pushdown automaton, on the other hand, reads data sequentially, and can thus be used to process XML documents more efficiently in a *streaming* fashion, as proposed by Kumar *et al.* [121].

In this context, learning can be used to obtain a specification (in the form of a visibly pushdown automaton) in situations where there is no DTD or schema available. For example, a legacy program or a web service might parse and validate input XML documents programmatically, and apply validation rules which are nowhere (formally) stated. By generating XML documents to be fed to this program, a learning algorithm would then infer the *structure* that valid documents need to possess, and the generated visibly pushdown automaton could be used for the off-line validation of documents, or also to obtain a formal description such as an XML schema or a DTD.

Compositional verification for pushdown systems. The second prototypical application of visibly pushdown languages is the verification of recursive programs. Again, the assumption that calls and returns are marked as such is not a real constraint in this domain. For the logical specification of such programs, Alur *et al.* [13] have proposed a temporal logic called CARET. Applications of visibly pushdown languages in the context of white-box program analysis are plenty: Chaudhuri and Alur [46], and also Roşu *et al.* [156] proposed basing monitors on visibly pushdown automata for respecting the procedural structure of calls and returns, and a general framework for temporal reasoning for procedural programs is presented by Alur and Chaudhuri [10].

[10] It is common to assume well-formedness, which means that a single return symbol—representing the closing tag for the topmost open one—is sufficient.

It is not quite clear how these results could translate to a black-box setting. Kumar *et al.* [119] introduce conformance testing of recursive Boolean programs, where the program is treated as a black-box that needs to be validated against a specification. However, while it is possible to instrument programs to make calls and returns visible as *output* (which could be useful for *passively* learning the specification of such a program from traces), it is hard to imagine how function calls and returns could be treated as inputs: when invoking a procedure, the subsequent or recursive invocation of other procedures is generally under the control of the program—possibly depending on further user input, which would however mostly correspond to *internal* actions; a granularity of the (black-box) alphabet that allows triggering the invocation of other procedures in arbitrary contexts is unlikely.

However, a fruitful application of VPA learning, which we leave as a direction for future research, could be assumption learning in *compositional verification*, as originally proposed by Cobleigh *et al.* [57]. Compositional verification is one proposed attempt to tackle the so-called *state-space explosion problem* [56], i.e., the problem that the state-space of a complex system composed of several components usually is far too big to be handled by an explicit-state model checker. The approach of compositional verification is therefore to reduce the verification of the whole system to verifying its components. However, components typically can only be proven to work correct in a certain *environment*, which is given by the remaining components. Assume-guarantee reasoning [151] aims at substituting the environment with an *assumption* about its behavior, which is much more abstract (i.e., smaller) than the actual environment, but precise enough to allow the analyzed component to guarantee correct behavior.

Since formulating such assumptions manually is challenging, Cobleigh *et al.* [57] proposed to *learn* them using active automata learning. Naturally, this limits the assumptions—and thus also the complexity of systems that can be analyzed—to whatever the learning algorithm can produce. The original approach therefore focused on safety properties, and Farzan *et al.* [65] extended this approach to *liveness properties* assumptions by presenting a learning algorithm for arbitrary ω-regular languages. The VPL learning algorithm developed in this chapter thus can be used to extend this approach to subclasses of context-free properties as assumptions (e.g., corresponding to safety CARET formula). Due to the increased complexity of the composition operation for visibly pushdown systems, the benefit of learning an assumption much smaller than the respective environment would be even greater.

7. Related Work

In this chapter, we will discuss other works that are related to this thesis. We will first focus on works that are directly relevant to the contents of this thesis, but also discuss works that, while not being related to the core subjects of this thesis, are still related to the overall subject of active automata learning, allowing the reader to obtain an overview.

7.1. Works Directly Related to the Contents of This Thesis

The next subsections discuss works that are similar or related to the contents of the previous chapters, in the order in which they occur in this thesis. There are three lines of works that we consider to be directly relevant to the contents of this thesis: approaches for unifying and formalizing the description of active automata learning algorithms (Chapter 3), algorithmic improvements to *classical* active DFA learning (Chapters 4 and 5), and active learning of (visibly) pushdown languages (Chapter 6). More general advancements, such as for richer classes of languages that are however not related to visibly pushdown languages, will be discussed in Section 7.2.

7.1.1. Unifying Formalization of Active Automata Learning

The need for a unifying and more formal description of active automata learning algorithms has been identified by a handful of other researchers. The best-known attempt at this is probably the *observation pack* framework due to Balcázar *et al.* [25], with the stated goal of providing a unified view on the learning algorithms due to Angluin [19], Rivest and Schapire [154, 155], and Kearns and Vazirani [115]. In their framework, an *observation pack* is a family of *observations* (i.e., examples with the observed output value), which are organized in a certain way and need to satisfy certain properties for being able to construct an automaton from them.

A formalization that bears many similarities to aspects of the framework presented in Chapter 3 of this thesis was presented by Berg *et al.* [31], motivated by the aim of unifying the descriptions of active automata learning and *conformance testing* [38]. The authors introduce the concept of suffix-observability, that the descriptions in this thesis also rely on to ensure an easy transfer to Mealy machines. Furthermore, there are some formal similarities: the concept of an *observation structure* is introduced, which is a partial function mapping words (elements of $\mathcal{U} \cup \mathcal{U}\Sigma$) to partial functions from Σ^* to $\mathcal{D}$, which is very similar to our notion of black-box classifiers and abstractions.

The main differences between the framework developed in Chapter 3 of this thesis and the two above-mentioned attempts is that both of them encode a number of assumptions in their formalization, that then result in a loss of generality. Examples for such assumptions are that representative short prefixes are unique, form a prefix-closed set, that there is a global set of suffixes, or that this set is suffix-closed. Imposing such assumptions of course makes it impossible

to identify certain phenomena (e.g., reachability or output inconsistencies), the characterization and precise description of which is one of the main results of Chapter 3. Also, the important subject of counterexample analysis is only addressed very briefly, if at all.

7.1.2. Algorithmic Improvements of Classical Active Automata Learning

Since Anlguin's initial presentation of L* [19], only a handful of improved versions or novel algorithms have been presented. Rivest and Schapire [154, 155] introduced the idea of using binary search to determine a single suffix of a counterexample that causes refinement, while Kearns and Vazirani [115] suggested replacing the observation table with a discrimination tree. Howar [93] then combined both ideas, resulting in the Observation Pack algorithm.

Counterexample handling. Maler and Pnueli [127] proposed the strategy of adding all suffixes of a counterexample to the table (commonly referred to as L^*_{col}). Irfan *et al.* [105, 106] observed the impact of long counterexamples on the number of membership queries, and presented the Suffix1by1 heuristic to reduce the number of suffixes. However, suffix-closedness is maintained at the cost of adding unnecessary suffixes, resulting in a worst-case query complexity that grows linearly with the counterexample length (whereas Rivest and Schapire's approach only results in a logarithmic growth; cf. also Table A.1).

Further addressing the problem of long counterexamples, Isberner and Steffen [108] proposed to use a binary search strategy also for Kearns and Vazirani's algorithm, and to furthermore use exponential search to avoid the problem that binary search poses at least one query of length $m/2$, while maintaining a logarithmic worst-case complexity.

Another approach to tackling long counterexamples has been presented by Aarts [1], adapting a technique proposed by Koopman *et al.* [117] for shortening counterexamples in the context of model-based testing: the current hypothesis is used to heuristically detect possible loops in the counterexample, which are then eliminated. As the check for "true" cycles are cheap to execute, significant benefits can be obtained in practice.

However, all of these techniques are of heuristic nature. They may work very well in most practical circumstances, but there might always be pathological cases where they fail to reduce the length of the counterexample significantly. On the other hand, TTT might require more hard sifts for finalizing discriminators, but after a refinement step is complete, the resulting impact on the internal data structures is the same as if a minimal counterexample had been processed. As our experiments furthermore indicate that there is no noticeable overhead when processing minimal counterexamples, some of the above techniques (such as cycle removal) can furthermore be combined with TTT to (heuristically) improve the average-case efficiency, but nonetheless limit the negative impact in pathological cases.

Space complexity. Merten *et al.* [138, 140] addressed the issue of space complexity of learning algorithms with their presentation of the DHC algorithm, motivated by the fact that the improved implementation quality of learning algorithms allowed to learn systems of such size that space consumption became a real problem. However, the favorable space complexity of DHC comes at the cost of not storing its observations. That is, in every refinement step, all previously asked queries have to be asked again. This can be remedied by using a *cache*, which however increases the space consumption again. Furthermore, the DHC algorithm does not minimize the suffixes extracted from a counterexample, meaning its space complexity depends on m and is not truly linear in the size of the hypothesis.

Other algorithms and approaches. Meinke *et al.* [132, 133] proposed several algorithms (CGE, ICGE) for learning-based testing [136] scenarios, which are based on string rewriting and universal algebra [135]. These algorithms follow an approach that can be considered as being dual to the algorithms considered in this thesis: instead of refining an approximation of the Nerode congruence, they start with a maximally fine relation and subsequently join classes. The authors report superior performance for learning-based testing applications.

Bollig *et al.* [36] presented an observation table-based algorithm, called NL*, which learns a certain class of NFAs, namely RSFAs [62]. Since NFAs and DFAs are equi-expressive, this does not extend the range of their learning algorithm beyond the scope of regular languages, but merely concerns representation. However, RSFAs may be exponentially more succinct than equivalent canonical DFAs, and the authors report favorable performance on a certain set of benchmarks (consisting of randomly-generated regular expression), however only comparing their algorithm to L* and L^*_{col}. Another approach for learning NFAs of was presented by Björklund *et al.* [35]. They describe an observation table-based algorithm for learning *universal automata* [83, 124], a certain class of NFAs that are based on the concept of *factors* of a language, without reporting on experimental results.

7.1.3. Extending Active Automata Learning to Context-Free Structures

In her paper presenting L*, Angluin [19] describes a possible modification for learning context-free grammars in Chomsky normal form, however requiring the learner to know the non-terminals (corresponding to the states in a pushdown automaton, and learning only the transitions [119]). Angluin and Kharitonov [21] prove that context-free languages can in general not be learned from a MAT alone. It has often been pointed out (e.g., by Clark [53]) that a MAT answering equivalence queries cannot even exist in a white-box setting, as equivalence of two context-free grammars is undecidable. For this reason, most approaches focus on passive learning of (subclasses of) context-free grammars from positive [54, 157] or positive and negative examples [74]. Some approaches combine positive examples with membership queries, yielding polynomial-time algorithms [55].

Alur and Madhusudan [11] relate visibly pushdown languages to the regular tree language of *stack trees*. Kumar *et al.* [119] point out that it would be possible to combine this result with the algorithm due to Sakakibara [157] (or the improved version by Drewes and Högberg [64]) for learning regular tree languages to obtain a tree language representation of a visibly pushdown language, which might however be non-deterministic and furthermore not exhibit certain structural properties to be expected from recursive programs.

Neider and Löding [143] investigate learning of visibly one-counter automata, which form a strict subclass of visibly pushdown automata, in a MAT-like setting. Their approach is based on learning a regular structure in the infinite *behavior graph*, using a data structure called "stratified observation table", and requiring access to a modified form of equivalence queries that permit restricting the subset of the target language on which equivalence is checked.

A proper active learning algorithm for VPAs was first described by Kumar *et al.* [119] for *complete modular* VPAs. The description is however very brief, and mostly relies on the reader's intuition to transfer the concepts of classical active automata learning to the setting of VPAs. The accompanying technical report [120] provides significantly more details. However, a formal framework that would allow to reason about other approaches (e.g., suffix-based instead

of prefix-based counterexample analysis) or a more detailed discussion of the properties maintained by the algorithm is not established. As their algorithm requires a number of membership queries that is at least linear in the length of a counterexample (which, in turn, may be exponential in the number of locations), the assumption of a *cooperative teacher* is proposed: by representing the counterexample in a certain, compact way (i.e., as a recursive equation system), the query complexity could be kept sub-exponentially. In contrast, leveraging the counterexample analysis framework developed in Section 3.3 of this thesis allows us to describe a counterexample analysis that is logarithmic in the length of the counterexample, and thus requires a polynomial number of queries for arbitrary counterexamples of (single) exponential length.

7.2. Other Works Related to Active Automata Learning

In this section, we will discuss works that, while not being directly relevant to the contents of this thesis, represent important advancements in the field of automata learning, or are relevant for obtaining an understanding and overview of the context of the field.

7.2.1. Grammatical Inference and Passive Automata Learning

Active automata learning is a subfield of *grammatical inference* [61] (sometimes also called *grammar induction*), which is concerned with inferring ("learning") formal descriptions of languages (such as grammars or automata), including probabilistic languages or transducers which are not formal languages in the *strict* sense (i.e., described by a formal grammar as defined by Chomsky [51]). Grammatical inference itself has its origins in computational linguistics and pattern recognition, and is sometimes also related to machine learning [141].

Many approaches in grammatical inference can be described as *passive learning*, i.e., they construct automata (or other formal descriptions) from sets of examples. Gold [75] and Angluin [18] showed that computing the smallest DFA consistent with a given sample (i.e., a set of words labeled with 0 or 1) was NP-hard. This has led to the development of (polynomial-time) heuristics such as the RPNI algorithm due to Oncina and García [147], or the k-tails algorithm due to Biermann and Feldman [34], which do not guarantee inferring the *minimal* consistent DFA, but often return a sufficiently small one. Similar techniques for inferring transducers from sample sets have been developed, such as the OSTIA algorithm by Oncina *et al.* [148].

Passive automata learning approaches are often used in the context of *specification mining* [17], which attempts to automatically discover formal specifications of, e.g., protocols, by observing regular program executions (for instance through analysis of log files). The aim of specification mining, however, is to focus on *normal* behavior, whereas active automata learning aims at exploring *all* possible behaviors.

A mixture between passive and active learning is *inductive testing*, as proposed by Walkinshaw *et al.* [170]: an initial set of traces is used to construct a model, which then forms the basis for generating test-cases. Executing these test-cases augments the sample set, allowing to construct a refined model.

7.2.2. Extending Active Automata Learning Beyond Regular Languages

Maler and Pnueli [127] described an extension of the L* algorithm to a strict subclass of ω-

regular languages (Büchi automata), and Farzan *et al.* [65] later presented an algorithm for actively inferring arbitrary ω-regular languages.

An adaption of L* for Mealy machines was first presented by Niese Margaria *et al.* [129], Niese [146], and its description later formalized by Shahbaz and Groz [161]. Based on Mealy machine inference, Aarts and Vaandrager [2] have developed a technique for learning I/O automata [125] under some additional assumptions.

The above techniques have in common that they still infer models with an inherently finite structure. To address the problem of effectively infinite alphabets (e.g., network messages containing integer values), Aarts *et al.* [3] proposed the use of (manually supplied) abstractions. Howar *et al.* [95] proposed an approach for automated counterexample-guided black-box inference of a maximally coarsest alphabet abstraction (assuming a finite state-space but a potentially infinite alphabet), which was later improved by Isberner *et al.* [109] to cover state-local alphabets. A similar approach is due to Maler and Mens [126, 137], however assuming that alphabet symbols exhibit some properties (e.g., an ordering) on which their partitioning can be based.

A lot of effort has been spent on a more adequate handling of *data* in the context of active automata learning. Several formalisms have been proposed to model systems that may pass around data values from unbounded domains, such as register automata [43, 44] (which can be regarded as variants of the finite-memory automata introduced by Kaminski and Francez [114]) or scalar-set Mealy machines [5]. Learning algorithms for the first kind have initially been described by Howar *et al.* [98], with a later extension to Mealy machines [97]; a survey on these approaches, including the historical developments and highlighting important stepping stones, has been given by Isberner *et al.* [111]. The original constraint that data values could be tested for equality only was overcome in a recently presented extension [45] that relies on SMT solving.

Algorithms for learning scalar-set Mealy machines have initially been presented by Aarts *et al.* [5, 6]. Aarts [1], in her PhD thesis, describes an improved approach that overcomes many of the original limitations in comparison with the above approach of learning register automata. A comparison of both approaches was later conducted jointly [9].

The above work extends active automata learning from finite-state to infinite-state systems, by allowing a finite number of memory locations that can store data values ranging over an unbounded domain and thus inducing an infinite *configuration (state) space*. However, a key limitation is that the *control structure* remains finite. The work on inferring VPAs can thus be considered as being orthogonal to the above: in this setting, the control structure is context-free and thus inherently infinite, while data can only be abstracted in a finite manner (i.e., by encoding it into the alphabet and the locations, both of which are finite). An intuitive illustration of this is the inference of a stack data structure, as described by Isberner *et al.* [111]: using register automata learning, a stack with a *finite* capacity storing data values from an *infinite* domain can be learned. In contrast, using VPA learning it is possible to learn a stack with *infinite* capacity, that can however only store data values from a *finite* domain.

7.2.3. Applications of Active Automata Learning in Formal Methods

Model checking. The first publication proposing the combination of active automata learning and formal methods was "Black-box checking" by Peled *et al.* [149, 150]. The goal of checking

a system against a (temporal) specification is accomplished by learning an initial model of a system, and using a mixture of *model checking* [24, 56] (for either detecting true specification violations, or obtaining *spurious* counterexamples to be used for refining the model) and conformance testing [38, 39], e.g., using the Vasilevskii-Chow method [52, 169] for generating counterexamples. This work has spun off a lot of related approaches focusing on enabling model checking or other model-based testing techniques in the setting of inexistent or inadequate models. Groce *et al.* [81, 82], e.g., proposed *adaptive model checking*, also based on active automata learning, as a means to deal with inconsistencies between existing but incorrect models and the actual system.

Test-case generation. Hagerer, Hungar *et al.* [84, 85, 101, 102, 129] were the first to report on using active automata learning for model generation of realistic systems in a practical setting, initially focusing on CTI (computer telephony integration) systems. Models for legacy systems, which were generated by using automata learning, were used to re-organize and improve test suites. This scenario quickly inspired engineering efforts to improve the practical performance [103, 130, 131], e.g., by using optimizing filters, led to the design of algorithms better suited for reactive systems [129, 146], and furthermore motivated the development of dedicated active automata learning tools and libraries [152, 153].

Network systems and protocols. Active automata learning is often used in the context of network systems or protocols, or hardware systems, as these represent black-box systems in their purest form. Aarts *et al.* [3] proposed a technique to learn models of communication protocols, and subsequently conducted a number of case studies, inferring models of, e.g., the EU biometric passport [4], bank cards [8], or the bounded retransmission protocol [7]. The technique has also been used successfully by Fiterău-Broştean *et al.* [66] for learning fragments of the TCP protocol.

Related is the use of active automata learning in the CONNECT project [28, 76, 113], which focused on ensuring interoperability of networked components in heterogeneous environments. Here, learning was used to infer state-machine models of networked systems [30], and these models were subsequently used to synthesize connectors realizing interoperability [29].

Security. Cho *et al.* [48] presented an application of active automata learning in the security space that received a lot of attention: using L^*_M [129, 146, 161], they inferred a formal model of the command and control protocol of a botnet. This model exhibited characteristics of the protocol that could be used to devise a takedown strategy.

MACE [49] is another application in the security domain, which uses active automata learning to recognize vulnerabilities in network protocol implementations. However, the proposed approach is a white-box one: learning is combined with symbolic or concolic execution [73, 116] and assists the latter by identifying the behavioral structure on a larger scale.

Interface synthesis. Alur *et al.* [16] presented an automata learning-based approach to generating temporal interfaces for Java classes (i.e., automata recognizing the language of safe sequences of operations on objects). Predicate abstraction [77] is used to obtain an abstracted version of the class, and the L^* algorithm is then used to synthesize a corresponding model. A similar approach was later proposed by Giannakopoulou *et al.* [71, 99], however based on symbolic execution instead of predicate abstraction, allowing the inference and automated refinement of symbolic transition guards.

Compositional verification. Cobleigh *et al.* [57] proposed an approach to compositional verification based on active automata learning: the state-explosion problem, due to multiple components acting in parallel, is circumvented by means of assume-guarantee reasoning. That is, the problem of checking whether the composed system of two components conforms to a specification is reduced to checking whether one of the component behaves correctly under a certain *assumption* (that is more abstract than the concrete behavior of the other component), and whether the other component guarantees this assumption. Active automata learning is used to *learn* such an assumption. The original approach worked for safety properties only, Farzan *et al.* [65] presented an extension admitting any ω-regular property to be learned as an assumption. A symbolic version was proposed by Alur *et al.* [15].

Active continuous quality control. This technique, introduced by Windmüller *et al.* [145, 171, 172], aims at providing a better understanding of the *evolution* of a software system over time. By inferring and then comparing models of different versions of a product (e.g., different releases or revisions in a source code management system), developers can inspect graphical visualizations of functional changes, to better understand the implications of intended ones, and to recognize unintended ones. Combined with checking the models against a specification, error traces detected in one version can be used as counterexamples to refine existing models of previous versions, allowing to precisely pinpoint when a certain behavioral change was introduced.

7.2.4. Active Automata Learning Tools and Framework

The plethora of practical applications of active automata learning calls for tools and libraries that offer these functionalities in a ready-to-use and reusable fashion. While the L* algorithm [19] is relatively easy to implement, this is not true for more sophisticated algorithms, such as the ones presented in this thesis. The same holds for many—often thoroughly engineered—optimizations concerning the practical performance [130, 131]. As learning often takes considerable time (Cho *et al.* [48] report a period of three weeks for inferring the botnet protocol), an existing, well-engineered and highly optimized learning algorithm implementation should therefore always be preferred over a—almost inevitably much simpler and less optimized—one-off implementation of a standard learning algorithm such as L*.

LearnLib[1] [152, 153] was probably the first active automata learning framework. Originally written in C++ and not being publicly available, the current version [112], developed by the author and others, is now based on Java and released under an open-source license. *LearnLib* features most active automata learning algorithms that have been described in the literature (including the TTT algorithm), a rich infrastructure, and high scalability.

libalf[2] [37] is an open-source automata learning framework written in C++. It focuses entirely on the algorithmic part and does not provide further infrastructure, e.g., to connect to real-life systems. In contrast to *LearnLib*, it also features some passive learning algorithms such as RPNI [147].

Tomte[3] [1] is a tool focusing on the automated inference of abstractions required to learn realistic systems. It uses learning algorithms from *LearnLib* internally, and complements these with simultaneously inferred stateful abstractions, as described by Aarts *et al.* [5, 6].

[1] http://www.learnlib.de/
[2] http://libalf.informatik.rwth-aachen.de/
[3] http://tomte.cs.ru.nl/

8. Conclusions

In this thesis, we have addressed the problem of *active automata learning*, i.e., learning finite-state machine models by experimentation, from a theoretical and algorithmic perspective. As a result, we have described a novel mathematical framework allowing to reason about how and why active automata learning algorithms work. The attained insights have led to the development of a new, highly efficient active automata learning algorithm that outperforms virtually every other algorithm in the presence of long counterexamples. The ideas underlying this algorithm could furthermore be translated with only little modification to the more complex scenario of visibly pushdown automata.

In Section 1.1, we have formulated three *research questions* that we wanted to address in this thesis. The first one concerned the formalization of active automata learning:

> *How can the phenomena encountered in active automata learning be characterized formally and independently of a concrete algorithmic realization, what is their significance, and what are desirable properties and characteristics that a learning algorithm should possess?*

The purely mathematical formulation of the framework presented in Chapter 3 allowed us to characterize two desirable semantic properties—reachability and output consistency—along with syntactic properties guaranteeing them: prefix-closedness and (semantic) suffix-closedness. As most early active automata learning algorithms actually enforced these syntactic properties, the inconsistencies resulting from a relaxation were only described later, and not truly as independent phenomena. While Lee and Yannakakis [122] observed that forgoing suffix-closedness, as proposed by Rivest and Schapire [155], leads to non-canonical hypotheses, Steffen *et al.* [167] as well as Van Heerdt [86] additionally observed that in this case the information in the observation table must be inconsistent with the hypothesis, and thus gives rise to a counterexample. The proposed strategy, however, was to simply analyze this counterexample. The same is true for the "dual" phenomenon of reachability inconsistencies, which can be observed during runs of Kearns and Vazirani's algorithm [115].

Our formalization yields the important insight that it is more adequate to reverse the perspective: counterexamples can themselves be regarded as either output or reachability inconsistencies. This allows a more efficient analysis, as a reachability inconsistency does not need to constitute a counterexample, and, conversely, analyzing an output inconsistency does not require the corresponding state to be reachable. Furthermore, we showed that both "counterexample" analysis strategies are applicable even when most "reasonable" assumptions—such as prefix-closedness of $\mathcal{U}$ or maintaining unique representatives—are dropped, and that they can both be reduced to a much simpler, abstract problem. This uncovers a beautiful symmetry between the role of prefix-based and suffix-based analysis (and of prefixes and suffixes in general), culminating in the observation that in both cases there is a direct correspondence between the immediate refinement suggested by the analysis result, and the violation of the corresponding syntactical property.

How can the insights gained through a rigorous formalization be translated into an efficient active learning algorithm, and how does the practical performance of an algorithm designed along these guidelines differ from existing algorithms?

Analyzing existing algorithms under the perspective of whether they manage to enforce the identified desirable properties of reachability and output consistency, it seemed that apparently one of them has to be sacrificed for maintaining the other. This is a direct consequence of the above observation on how the application of prefix- or suffix-based counterexample analysis violates prefix- or suffix-closedness, respectively. Pursuing the naturally arising question of whether it is possible to maintain both these syntactic properties simultaneously led to the development of the TTT algorithm, the idea of which can be summarized as follows: the observation that maintaining one property inevitably leads to temporary violations of the other gives rise to the approach of explicitly *restoring* the respective property as soon as possible, thus "purging" the internal data structures from the effect of non-minimal counterexamples.

The theoretical complexity analysis revealed that TTT is *space-optimal*, meaning that no other active automata learning algorithm can require an asymptotically lower amount of memory. This result reflects a very economic handling of information, which is of particular importance under the initially stated goal that an ideal learning algorithm should ask only those questions that need to be asked.

Furthermore, while the theoretical query complexity analysis suggested that this results in no asymptotic improvements over some existing algorithms, the experimental evaluation painted a very different picture: in the case of long counterexamples, TTT is superior to all other algorithms, to the extent that the length of counterexamples has little to no performance impact. Notably, this does not only concern the number of symbols, which were our major concern when developing this algorithm, but also the number of queries. Since the cleaning up of data structures incurs no noticeable overhead in the presence of optimal counterexamples, we feel confident to state that the TTT algorithm, developed along the above guidelines, is indeed superior to every other algorithm in virtually all circumstances.

To what extent—and if so, how—can the mathematical formalization and the identified principles of efficient algorithm design be transferred to the active inference of richer classes of models, e.g., modeling infinite-state systems?

While the TTT algorithm for regular languages is already technically quite involved, the rigorous mathematical formalization allowed us to develop an extension for learning visibly pushdown automata in a comparably simple manner. Building on a congruence-based characterization of visibly pushdown languages developed by Alur *et al.* [14], most of the concepts from active learning of regular languages could be transferred in a relatively direct and very natural fashion. This in particular includes counterexample analysis, which can be dealt with as an instantiation of the abstract framework developed in Chapter 3, and thus be tackled using search algorithms of logarithmic worst-case complexity. This is of particular importance in the context of visibly pushdown systems, as counterexamples may be of exponential length even in the presence of a *cooperative teacher*.

An essential step for the TTT algorithm was the identification of *finalization rules*, based on an inductive characterization of inequivalence in the Nerode congruence. Identifying similar rules for the case of visibly pushdown systems provided a clear guideline to developing a variant

of TTT for visibly pushdown automata, named TTT-VPA. Compared to an algorithm without these finalization steps, TTT-VPA exhibits superior and in particular more *stable* performance, suggesting that the TTT approach of maintaining a minimal internal representation is indeed the key to developing robust and scalable automata learning algorithms.

8.1. Future Work and Open Problems

Despite the impressive pace in which active automata learning has evolved over the past years, there are still an enormous number of unsolved questions and rather poorly-understood phenomena to be found in the field. In the following, we will elaborate on some aspects that we could only touch upon in the scope of this thesis, or that can be regarded as a straightforward attempt of taking the results of this thesis to the next level.

Further investigation of the prefix/suffix symmetry. The mathematical framework developed in Chapter 3 uncovered remarkable symmetries between the role of prefixes and suffixes, especially concerning the two prevalent counterexample analysis strategies. The TTT algorithm developed in this thesis is based on suffix-based counterexample analysis. However, it can be adapted to use prefix-based analysis with relatively little effort: Kearns and Vazirani's algorithm builds a suffix-closed set of discriminators, but violates prefix-closedness of $\mathcal{U}$. It could therefore be used as a starting point for a prefix-based variant of TTT, which eventually maintains both properties by restoring prefix-closedness of the access sequences. The similarities to the suffix-based version of TTT might help to find a characterization of situations which favor either prefix-based or suffix-based analysis. Moreover, it could contribute to an understanding as to why prefix-based analysis seems to be inherently more complex than suffix-based analysis (according to the worst-case analysis), a question raised in Section 3.3.6.

Optimality of learning algorithms. For TTT, we could only prove space-optimality (cf. Section 5.3.2), but not optimality regarding the query or symbol complexities. While the query complexity of $\mathcal{O}(kn^2 + n\log m)$ is close to the theoretical lower bound of $\Omega(kn^2)$ proven by Balcázar *et al.* [25], it fails to meet it for excessively long counterexamples, i.e., of length $m = 2^{\omega(kn)}$. Lower bounds analysis is often performed by considering a teacher who maintains a set of possible target systems, and every membership and equivalence query forces him to reveal an output, narrowing down this set. While the learner can choose the membership queries (and choose when to make an equivalence query), the teacher can choose an output that minimizes the potential reduction of the number of target systems. One would need to investigate whether there exist (classes of) target systems, for which counterexamples of such length could be generated such that analyzing them provably requires a certain amount of effort (i.e., $\Omega(\log m)$ queries). On the other hand, no counterexamples of length greater than $m = 2^{\omega(k^n)}$ need to be considered, as k^n is a trivial upper bound for a learning algorithm based on exhaustive exploration. Alternatively, it might be possible to show that counterexamples of length $m = 2^{\omega(kn)}$ contain so much inherent redundancy that it is possible to shorten them to counterexamples of length $2^{\mathcal{O}(kn)}$ by using no more than $\mathcal{O}(kn)$ queries per counterexample, which would result in an optimal overall query complexity. For simplicity, one could start by assuming that n is known to the learner.

The symbol complexity of learning algorithms has only rarely been considered, and not at all in the context of lower bounds and optimality. However, as most queries of length $\omega(n)$ occur during, or as a result of, counterexample analysis, a better understanding of the role that coun-

terexamples play in the context of lower bounds (see above) is likely to be a prerequisite for an analysis that should yield meaningful results.

Practical performance of learning algorithms. We have noted that the query complexity of TTT is almost optimal. The same, however, can already be said about Rivest and Schapire's [155] algorithm, that has the same worst-case query complexity (cf. Table A.1). Tables 5.1 and 5.2 speak a significantly different language. This shows that the asymptotic worst-case analysis is of only limited value. Since reasoning about average-case complexities is hard—the problem of merely *estimating* the number of canonical DFAs for given n and k is very involved [63], and it is unclear how a "reasonable" distribution over these would look like, or how DFAs could be sampled according to this distribution—, a better understanding on what *makes* a system hard or expensive to learn is required.

It is, for example, known that automata as the one from Figure 4.3 (Howar [92] calls them "key lock automata"), where a long sequence of symbols is required to reach the final state, are hard to learn, whereas randomly generated DFAs often require surprisingly few queries [94]. This suggests a connection between the (average or maximum) distance between the initial and other states in the automaton on one hand, and the required number of queries on the other hand. It would be interesting to investigate whether other such characterizations of "hard to learn" or "easy to learn" DFAs can be established.

Theoretical and practical aspects of VPA *learning.* The transfer of the theory for active automata learning from regular to visibly pushdown systems, as outlined in Section 6.3, was surprisingly easy. An interesting aspect to investigate is whether this can also be said for, e.g., optimality analysis: what is the lower bound for learning a visibly pushdown system? An obstacle could be the fact that, while regular languages have a *natural* canonical representation whose parameters (i.e., the size of the canonical DFA, or, equivalently, the index of the Nerode congruence) allow reasoning about a lower bound, the same is not true for VPAs. The complexity analysis for TTT-VPA that we gave in Section 6.4.5 is parametric in the size of the canonical 1-SEVPA of the target system, but there could exist a k such that for a k-partition of Σ_{call}, the corresponding k-SEVPA (or even other models, admitting multiple entries) is exponentially smaller, and the teacher cannot make any assumptions about the internal hypothesis representation of the learner.

A practically relevant case study employing VPA learning is yet to be done. The experimental evaluation in Section 6.5 was very preliminary, and realistic scenarios might expose completely different characteristics. We have outlined two possible applications of VPA learning in Section 6.6, namely XML document processing and compositional verification. An obstacle in practice might again be the choice of a canonical form: are 1-SEVPAs, which may be exponentially larger than other types of VPAs [14], a suitable model for representing XML document structures and temporal assumptions? If not, what are good heuristics to determine the corresponding alphabet partitions in black-box scenarios? It is unlikely that these questions can be answered in general, which is why case studies are essential for assessing the feasibility in the first place.

Applying the TTT idea to other classes. We have described the TTT idea of "purging" the internal data structures to eliminate the impact of long counterexamples in detail for learning DFAs, Mealy machines, and visibly pushdown automata. The demonstrated efficacy suggests that it may yield fruitful benefits for other classes of systems as well. The most interesting example would certainly be register automata [1, 5, 98, 111], in particular since the number of queries typically grows exponentially with the length of counterexamples. Aarts *et al.* [9], in

their comparison of *LearnLib* [112, 139] and *Tomte* [1], report that, while the performance of both approaches is similar for optimal counterexamples, using a heuristic shortening of counterexamples [1, 117] as implemented in *Tomte* yields staggering benefits. Integrating the idea of even *minimizing* the distinguishing suffixes that are internally used for maintaining both the hypothesis structure and the abstraction[1] into the register automaton learning algorithm by Howar *et al.* [98] (or even the improved version relying on SMT-solving [45]) therefore sounds highly promising.

[1] *Tomte* uses a normal Mealy machine learning algorithm under the hood, and maintains the abstraction ("mapper") separately. Since the abstraction is also refined in a counterexample-guided fashion that is very similar to the learning process, applying the TTT idea on a full scale would also require modifications to *Tomte*, and is not accomplished merely by using TTT as the underlying Mealy machine learning algorithm.

Bibliography

[1] Fides Aarts. *Tomte: Bridging the Gap Between Active Learning and Real-World Systems*. PhD thesis, Radboud University Nijmegen, 2014. URL http://hdl.handle.net/2066/130428. (Cited on pages 1, 3, 148, 151, 153, 158, and 159.)

[2] Fides Aarts and Frits Vaandrager. Learning I/O Automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin / Heidelberg, 2010. DOI: 10.1007/978-3-642-15375-4_6. (Cited on page 151.)

[3] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, *Testing Software and Systems*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16572-6. DOI: 10.1007/978-3-642-16573-3_14. (Cited on pages 151 and 152.)

[4] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and Abstraction of the Biometric Passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16557-3. DOI: 10.1007/978-3-642-16558-0_54. (Cited on page 152.)

[5] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata Learning through Counterexample Guided Abstraction Refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32758-2. DOI: 10.1007/978-3-642-32759-9_4. (Cited on pages 1, 151, 153, and 158.)

[6] Fides Aarts, Faranak Heidarian, and Frits Vaandrager. A Theory of History Dependent Abstractions for Learning Interface Automata. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 240–255. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32939-5. DOI: 10.1007/978-3-642-32940-1_18. (Cited on pages 151 and 153.)

[7] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. Learning and Testing the Bounded Retransmission Protocol. In *Proceedings of the* 11^{th} *International Conference on Grammatical Inference*, volume 21 of *JMLR W&CP*, 2012. URL http://www.jmlr.org/proceedings/papers/v21/aarts12a/aarts12a.pdf. (Cited on page 152.)

[8] Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 461–468, March 2013. DOI: 10.1109/ICSTW.2013.60. (Cited on page 152.)

[9] Fides Aarts, Falk Howar, Harco Kuppens, and Frits Vaandrager. Algorithms for Inferring Register Automata. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 8802 of *Lecture Notes in Computer Science*, pages 202–219. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-45233-2. DOI: 10.1007/978-3-662-45234-9_15. (Cited on pages 151 and 158.)

[10] Rajeev Alur and Swarat Chaudhuri. Temporal Reasoning for Procedural Programs. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 45–60, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11318-4, 978-3-642-11318-5. DOI: 10.1007/978-3-642-11319-2_7. (Cited on page 144.)

[11] Rajeev Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM. ISBN 1-58113-852-0. DOI: 10.1145/1007352.1007390. (Cited on pages 4, 118, 119, 144, and 149.)

[12] Rajeev Alur and P. Madhusudan. Adding Nesting Structure to Words. *J. ACM*, 56(3):16:1–16:43, May 2009. ISSN 0004-5411. DOI: 10.1145/1516512.1516518. (Cited on page 144.)

[13] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A Temporal Logic of Nested Calls and Returns. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21299-7. DOI: 10.1007/978-3-540-24730-2_35. (Cited on page 144.)

[14] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for Visibly Pushdown Languages. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1102–1114. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27580-0. DOI: 10.1007/11523468_89. (Cited on pages 123, 124, 125, 156, and 158.)

[15] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic Compositional Verification by Learning Assumptions. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27231-1. DOI: 10.1007/11513988_52. (Cited on page 153.)

[16] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the* 32^{nd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 98–109, New York, NY, USA,

2005. ACM. ISBN 1-58113-830-X. DOI: 10.1145/1040305.1040314. (Cited on pages 1 and 152.)

[17] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. DOI: 10.1145/503272.503275. (Cited on page 150.)

[18] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978. ISSN 0019-9958. DOI: 10.1016/S0019-9958(78)90683-6. (Cited on page 150.)

[19] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. DOI: 10.1016/0890-5401(87)90052-6. (Cited on pages 1, 2, 3, 25, 29, 30, 46, 49, 51, 57, 58, 77, 89, 110, 118, 147, 148, 149, 153, and 181.)

[20] Dana Angluin. Queries and Concept Learning. *Mach. Learn.*, 2(4):319–342, April 1988. ISSN 0885-6125. DOI: 10.1023/A:1022821128753. (Cited on page 25.)

[21] Dana Angluin and Michael Kharitonov. When Won't Membership Queries Help? In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 444–454, New York, NY, USA, 1991. ACM. ISBN 0-89791-397-3. DOI: 10.1145/103418.103420. (Cited on page 149.)

[22] Dana Angluin, Lisa Hellerstein, and Marek Karpinski. Learning Read-once Formulas with Queries. *J. ACM*, 40(1):185–210, January 1993. ISSN 0004-5411. DOI: 10.1145/138027.138061. (Cited on page 4.)

[23] Peter R.J. Asveld and Anton Nijholt. The inclusion problem for some subclasses of context-free languages. *Theoretical Computer Science*, 230(1–2):247–256, 2000. ISSN 0304-3975. DOI: 10.1016/S0304-3975(99)00113-9. (Cited on page 118.)

[24] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X, 9780262026499. (Cited on pages 1, 7, and 152.)

[25] José L. Balcázar, Josep Díaz, Ricard Gavaldà, and Osamu Watanabe. Algorithms for Learning Finite Automata from Queries: A Unified View. In Ding-Zhu Du and Ker-I Ko, editors, *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Springer US, 1997. ISBN 978-1-4613-3396-8. DOI: 10.1007/978-1-4613-3394-4_2. (Cited on pages 2, 27, 51, 57, 84, 147, and 157.)

[26] Borja Balle. Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries. Technical report, Departament De Llenguatges I Sistemes, Universitat Politècnica de Catalunya, Barcelona, 2010. URL http://www.cs.upc.edu/~bballe/papers/zulu10.pdf. (Cited on page 58.)

[27] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing System States by Active Learning Algorithms. In Alessandro Moschitti and Riccardo Scandariato, editors, *Eternal Systems*, volume 255 of *CCSE*, pages 61–78. Springer-Verlag, 2012. DOI: 10.1007/978-3-642-28033-7_6. (Cited on page 2.)

[28] Amel Bennaceur, Paola Inverardi, Valérie Issarny, and Romina Spalazzese. Automated Synthesis of CONNECTors to support Software Evolution. *ERCIM News*, 2012(88), 2012. URL http://ercim-news.ercim.eu/en88/special/automated-synthesis-of-connectors-to-support-software-evolution. (Cited on page 152.)

[29] Amel Bennaceur, Chris Chilton, Malte Isberner, and Bengt Jonsson. Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods - 11th International Conference (SEFM 2013)*, volume 8137 of *LNCS*, pages 274–288. Springer, 2013. DOI: 10.1007/978-3-642-40561-7_19. (Cited on page 152.)

[30] Amel Bennaceur, Valérie Issarny, Daniel Sykes, Falk Howar, Malte Isberner, Bernhard Steffen, Richard Johansson, and Alessandro Moschitti. Machine Learning for Emergent Middleware. In Alessandro Moschitti and Barbara Plank, editors, *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, volume 379 of *Communications in Computer and Information Science*, pages 16–29. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45259-8. DOI: 10.1007/978-3-642-45260-4_2. (Cited on page 152.)

[31] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25420-1. DOI: 10.1007/978-3-540-31984-9_14. (Cited on pages 19 and 147.)

[32] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's Learning. *Electron. Notes Theor. Comput. Sci.*, 118:3–18, February 2005. ISSN 1571-0661. DOI: 10.1016/j.entcs.2004.12.015. (Cited on page 110.)

[33] Antonia Bertolino, Antonello Calabrò, Maik Merten, and Bernhard Steffen. Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring. *ERCIM News*, 2012(88):28–29, 2012. URL http://ercim-news.ercim.eu/en88/special/never-stop-learning-continuous-validation-of-learned-models-for-evolving-systems-through-monitoring. (Cited on pages 87 and 88.)

[34] Alan W. Biermann and Jerome A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972. ISSN 0018-9340. DOI: 10.1109/TC.1972.5009015. (Cited on page 150.)

[35] Johanna Björklund, Henning Fernau, and Anna Kasprzik. MAT Learning of Universal Automata. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 7810 of *Lecture Notes in Computer Science*, pages 141–152. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37063-2. DOI: 10.1007/978-3-642-37064-9_14. (Cited on pages 59 and 149.)

[36] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style Learning of Nfa. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 1004–1009, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. URL http://ijcai.org/papers09/Papers/IJCAI09-170.pdf. (Cited on pages 46, 59, and 149.)

[37] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and DavidR. Piegdon. libalf: The Automata Learning Framework. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14294-9. DOI: 10.1007/978-3-642-14295-6_32. (Cited on page 153.)

[38] Ed Brinksma. Formal Methods for Conformance Testing: Theory Can Be Practical. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 44–46. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66202-0. DOI: 10.1007/3-540-48683-6_6. (Cited on pages 147 and 152.)

[39] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540262784. DOI: 10.1007/b137241. (Cited on pages 1 and 152.)

[40] Nader H. Bshouty. Exact Learning Boolean Functions via the Monotone Theory. *Information and Computation*, 123(1):146–153, 1995. ISSN 0890-5401. DOI: 10.1006/inco.1995.1164. (Cited on page 4.)

[41] Olaf Burkart and Bernhard Steffen. Model Checking for Context-free Processes. In W. R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer Berlin Heidelberg, 1992. ISBN 978-3-540-55822-4. DOI: 10.1007/BFb0084787. (Cited on page 118.)

[42] Olaf Burkart and Bernhard Steffen. Pushdown processes: Parallel composition and model checking. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-58329-5. DOI: 10.1007/BFb0015001. (Cited on page 118.)

[43] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A Succinct Canonical Register Automaton Model. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24371-4. DOI: 10.1007/978-3-642-24372-1_26. (Cited on pages 117 and 151.)

[44] Sofia Cassel, Bengt Jonsson, Falk Howar, and Bernhard Steffen. A Succinct Canonical Register Automaton Model for Data Domains with Binary Relations. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 57–71. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33385-9. DOI: 10.1007/978-3-642-33386-6_6. (Cited on page 151.)

[45] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning Extended Finite State Machines. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, volume 8702 of *Lecture Notes in Computer Science*, pages 250–264. Springer International Publishing, 2014. ISBN 978-3-319-10430-0. DOI: 10.1007/978-3-319-10431-7_18. (Cited on pages 1, 151, and 159.)

[46] Swarat Chaudhuri and Rajeev Alur. Instrumenting C Programs with Nested Word Monitors. In Dragan Bošnački and Stefan Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 279–283. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73369-0. DOI: 10.1007/978-3-540-73370-6_20. (Cited on page 144.)

[47] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning Minimal Separating DFA's for Compositional Verification. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-00767-5. DOI: 10.1007/978-3-642-00768-2_3. (Cited on page 52.)

[48] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 426–439, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. DOI: 10.1145/1866307.1866355. (Cited on pages 1, 2, 152, and 153.)

[49] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association. URL http://www.usenix.org/events/sec11/tech/full_papers/Cho.pdf. (Cited on page 152.)

[50] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. DOI: 10.1145/2509136.2509552. (Cited on page 27.)

[51] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956. ISSN 0096-1000. DOI: 10.1109/TIT.1956.1056813. (Cited on pages 21 and 150.)

[52] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. on Software Engineering*, SE-4(3):178–187, May 1978. ISSN 0098-5589. DOI: 10.1109/TSE.1978.231496. (Cited on pages 85 and 152.)

[53] Alexander Clark. Distributional Learning of Some Context-free Languages with a Minimally Adequate Teacher. In *Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications*, ICGI'10, pages 24–37,

Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15487-5, 978-3-642-15487-4. DOI: 10.1007/978-3-642-15488-1_4. (Cited on page 149.)

[54] Alexander Clark and Rémi Eyraud. Polynomial Identification in the Limit of Substitutable Context-free Languages. *J. Mach. Learn. Res.*, 8:1725–1745, December 2007. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1314498.1314556. (Cited on page 149.)

[55] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A Polynomial Algorithm for the Inference of Context Free Languages. In Alexander Clark, François Coste, and Laurent Miclet, editors, *Grammatical Inference: Algorithms and Applications*, volume 5278 of *Lecture Notes in Computer Science*, pages 29–42. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88008-0. DOI: 10.1007/978-3-540-88009-7_3. (Cited on page 149.)

[56] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999. (Cited on pages 1, 145, and 152.)

[57] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning Assumptions for Compositional Verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00898-9. DOI: 10.1007/3-540-36577-X_24. (Cited on pages 1, 145, and 153.)

[58] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An Interactive Learning Competition. In Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce Watson, editors, *Finite-State Methods and Natural Language Processing*, volume 6062 of *Lecture Notes in Computer Science*, pages 139–146. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14683-1. DOI: 10.1007/978-3-642-14684-8_15. (Cited on pages 2, 87, and 113.)

[59] Rene De La Briandais. File Searching Using Variable Length Keys. In *Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. ACM. DOI: 10.1145/1457838.1457895. (Cited on page 73.)

[60] Colin de la Higuera. A Bibliographical Study of Grammatical Inference. *Pattern Recogn.*, 38 (9):1332–1348, September 2005. ISSN 0031-3203. DOI: 10.1016/j.patcog.2005.01.003. (Cited on page 1.)

[61] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010. ISBN 0521763169, 9780521763165. (Cited on pages 1, 16, 25, 117, and 150.)

[62] François Denis, Aurélien Lemay, and Alain Terlutte. Residual Finite State Automata. *Fundamenta Informaticae*, 51(4):339–368, 2002. ISSN 0169-2968. (Cited on pages 59 and 149.)

[63] Michael Domaratzki, Derek Kisman, and Jeffrey Shallit. On the number of distinct languages accepted by finite automata with n states. *Journal of Automata, Languages and Combinatorics*, 7(4):469–486, 2002. (Cited on pages 106 and 158.)

[64] Frank Drewes and Johanna Högberg. Learning a Regular Tree Language from a Teacher. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, volume 2710 of *Lecture Notes in Computer Science*, pages 279–291. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40434-7. DOI: 10.1007/3-540-45007-6_22. (Cited on page 149.)

[65] Azadeh Farzan, Yu-Fang Chen, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_2. (Cited on pages 145, 151, and 153.)

[66] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning Fragments of the TCP Network Protocol. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 78–93. Springer International Publishing, 2014. ISBN 978-3-319-10701-1. DOI: 10.1007/978-3-319-10702-8_6. (Cited on page 152.)

[67] Emily P. Friedman. The inclusion problem for simple languages. *Theoretical Computer Science*, 1(4):297–316, 1976. ISSN 0304-3975. DOI: 10.1016/0304-3975(76)90074-8. (Cited on page 118.)

[68] Markus Frohme. Active Automata Learning with Adaptive Distinguishing Sequences. MSc thesis, TU Dortmund University, Dortmund, Germany, 2015. (Cited on page 59.)

[69] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19834-2. DOI: 10.1007/978-3-642-19835-9_33. (Cited on page 110.)

[70] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-Guarantee Verification of Source Code with Design-Level Assumptions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 211–220. IEEE Computer Society, May 2004. ISBN 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317443. (Cited on page 1.)

[71] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic Learning of Component Interfaces. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 248–264, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33124-4. DOI: 10.1007/978-3-642-33125-1_18. (Cited on pages 1 and 152.)

[72] Seymour Ginsburg and Sheila Greibach. Deterministic context free languages. *Information and Control*, 9(6):620–648, 1966. ISSN 0019-9958. DOI: 10.1016/S0019-9958(66)80019-0. (Cited on page 118.)

[73] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. DOI: 10.1145/1065010.1065036. (Cited on page 152.)

[74] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. ISSN 0019-9958. DOI: 10.1016/S0019-9958(67)91165-5. (Cited on page 149.)

[75] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. ISSN 0019-9958. DOI: 10.1016/S0019-9958(78)90562-4. (Cited on pages 1, 46, and 150.)

[76] Paul Grace, Nikolaos Georgantas, Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Valérie Issarny, Massimo Paolucci, Rachid Saadi, Bertrand Souville, and Daniel Sykes. The CONNECT Architecture. In *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, pages 27–52, 2011. DOI: 10.1007/978-3-642-21455-4_2. (Cited on page 152.)

[77] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63166-8. DOI: 10.1007/3-540-63166-6_10. (Cited on page 152.)

[78] Sheila Greibach. A note on undecidable properties of formal languages. *Mathematical systems theory*, 2(1):1–6, 1968. ISSN 0025-5661. DOI: 10.1007/BF01691341. (Cited on page 118.)

[79] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Inference of Timed Transition Systems. *Electronic Notes in Theoretical Computer Science*, 138(3):87–99, 2005. DOI: 10.1016/j.entcs.2005.02.062. (Cited on page 1.)

[80] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010. ISSN 0304-3975. DOI: 10.1016/j.tcs.2010.07.008. (Cited on page 1.)

[81] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive Model Checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43419-1. DOI: 10.1007/3-540-46002-0_25. (Cited on pages 1, 85, and 152.)

[82] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive Model Checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006. DOI: 10.1093/jigpal/jzl007. (Cited on pages 85 and 152.)

[83] Igor Grunsky, Oleksiy Kurganskyy, and Igor Potapov. On a Maximal NFA Without Mergible States. In *Proceedings of the First International Computer Science Conference on Theory*

and Applications, CSR'06, pages 202–210, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34166-8, 978-3-540-34166-6. DOI: 10.1007/11753728_22. (Cited on pages 59 and 149.)

[84] Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001. (Cited on pages 1 and 152.)

[85] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model Generation by Moderated Regular Extrapolation. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306, pages 80–95. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43353-8. DOI: 10.1007/3-540-45923-5_6. (Cited on pages 1 and 152.)

[86] Gerco van Heerdt. Efficient Inference of Mealy Machines. BSc thesis, Radboud University Nijmegen, Nijmegen, NL, June 2014. URL http://www.cs.ru.nl/bachelorscripties/2014/Gerco_van_Heerdt___4167503___Efficient_Inference_of_Mealy_Machines.pdf. (Cited on pages 31, 54, 58, and 155.)

[87] Markus Holzer and Barbara König. On deterministic finite automata and syntactic monoid size. *Theoretical Computer Science*, 327(3):319–347, 2004. ISSN 0304-3975. DOI: 10.1016/j.tcs.2004.04.010. Developments in Language Theory. (Cited on page 124.)

[88] John E. Hopcroft. An n log n Algorithm for Minimizing States in a Finite Automaton. Technical report, Stanford University, Department of Computer Science, Stanford, CA, USA, January 1971. URL http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf. (Cited on pages 22 and 67.)

[89] John E. Hopcroft and Richard M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. Technical report, Department of Computer Science, Cornell University, December 1971. URL http://hdl.handle.net/1813/5958. (Cited on page 109.)

[90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1969. ISBN 020102988X. (Cited on page 52.)

[91] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001. ISBN 978-0-201-44124-6. (Cited on page 7.)

[92] Falk Howar. Inferenz Parametrisierter Moore-Automaten. MSc thesis, TU Dortmund University, Dortmund, Germany, 2009. (Cited on page 158.)

[93] Falk Howar. *Active Learning of Interface Programs*. PhD thesis, TU Dortmund University, 2012. URL http://hdl.handle.net/2003/29486. (Cited on pages 3, 50, 51, 61, 76, 78, 84, 85, 109, 110, 111, 148, and 181.)

[94] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010. ISBN 978-3-642-16557-3. DOI: 10.1007/978-3-642-16558-0_55. (Cited on pages 2, 77, 85, 87, 111, 113, and 158.)

[95] Falk Howar, Bernhard Steffen, and Maik Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin / Heidelberg, 2011. DOI: 10.1007/978-3-642-18275-4_19. (Cited on page 151.)

[96] Falk Howar, Oliver Bauer, Maik Merten, Bernhard Steffen, and Tiziana Margaria. The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 232–247. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-34781-8_18. (Cited on page 2.)

[97] Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring Semantic Interfaces of Data Structures. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34025-3. DOI: 10.1007/978-3-642-34026-0_41. (Cited on page 151.)

[98] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In Viktor Koncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin / Heidelberg, 2012. DOI: 10.1007/978-3-642-27940-9_17. (Cited on pages 151, 158, and 159.)

[99] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. Hybrid Learning: Interface Generation Through Static, Dynamic, and Symbolic Analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 268–279, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. DOI: 10.1145/2483760.2483783. (Cited on pages 1 and 152.)

[100] Falk Howar, Malte Isberner, and Bernhard Steffen. Tutorial: Automata Learning in Practice. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 8802 of *Lecture Notes in Computer Science*, pages 499–513. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-45233-2. DOI: 10.1007/978-3-662-45234-9_34. (Cited on pages 3 and 91.)

[101] Hardi Hungar and Bernhard Steffen. Behavior-based model construction. *International Journal on Software Tools for Technology Transfer*, 6(1):4–14, 2004. ISSN 1433-2779. DOI: 10.1007/s10009-004-0139-8. (Cited on pages 1 and 152.)

[102] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *Proceedings of the 2003 International Test Conference (ITC 2003)*, volume 1, pages 971–980, October 2003. DOI: 10.1109/TEST.2003.1271205. (Cited on pages 1 and 152.)

[103] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-Specific Optimization in Automata Learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. 15th Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, July 2003. DOI: 10.1007/978-3-540-45069-6_31. (Cited on pages 1, 2, and 152.)

[104] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976. ISSN 0020-0190. DOI: 10.1016/0020-0190(76)90095-8. (Cited on page 76.)

[105] Muhammad Naeem Irfan. *Analysis and optimization of software model inference algorithms*. PhD thesis, Université de Grenoble, Grenoble, France, September 2012. (Cited on pages 3, 51, 53, 58, 110, 148, and 181.)

[106] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. Angluin Style Finite State Machine Inference with Non-optimal Counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing*, MIIT '10, pages 11–19, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0147-3. DOI: 10.1145/1868044.1868046. (Cited on pages 50, 51, 53, 58, 110, 148, and 181.)

[107] Muhammad-Naeem Irfan, Roland Groz, and Catherine Oriat. Improving Model Inference of Black Box Components having Large Input Test Set. In *Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012, University of Maryland, College Park, USA, September 5-8, 2012*, pages 133–138, 2012. URL http://jmlr.org/proceedings/papers/v21/irfan12a/irfan12a.pdf. (Cited on page 53.)

[108] Malte Isberner and Bernhard Steffen. An Abstract Framework for Counterexample Analysis in Active Automata Learning. In *Proceedings of the 12th International Conference on Grammatical Inference*, volume 34 of *JMLR Workshop & Conference Proceedings*, pages 79–93, 2014. URL http://jmlr.org/proceedings/papers/v34/isberner14a.pdf. (Cited on pages 4, 32, 37, 45, 50, 76, 78, 111, 132, 148, and 181.)

[109] Malte Isberner, Falk Howar, and Bernhard Steffen. Inferring Automata with State-Local Alphabet Abstractions. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, volume 7871 of *LNCS*, pages 124–138, 2013. DOI: 10.1007/978-3-642-38088-4_9. (Cited on page 151.)

[110] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and ScottA. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer International Publishing, 2014. ISBN 978-3-319-11163-6. DOI: 10.1007/978-3-319-11164-3_26. (Cited on pages 5, 27, 87, 88, and 181.)

[111] Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014. ISSN 0885-6125. DOI: 10.1007/s10994-013-5419-7. (Cited on pages 1, 151, and 158.)

[112] Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer International Publishing, 2015. ISBN 978-3-319-21689-8. DOI: 10.1007/978-3-319-21690-4_32. (Cited on pages 110, 153, and 159.)

[113] Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *ICECCS*, pages 154–161. IEEE Computer Society, June 2009. DOI: 10.1109/ICECCS.2009.44. (Cited on page 152.)

[114] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994. ISSN 0304-3975. DOI: 10.1016/0304-3975(94)90242-9. (Cited on pages 117 and 151.)

[115] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4. (Cited on pages 2, 47, 50, 51, 57, 58, 61, 77, 87, 89, 110, 119, 147, 148, 155, and 181.)

[116] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. DOI: 10.1145/360248.360252. (Cited on page 152.)

[117] Pieter Koopman, Peter Achten, and Rinus Plasmeijer. Model-Based Shrinking for State-Based Testing. In Jay McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, pages 107–124. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-45339-7. DOI: 10.1007/978-3-642-45340-3_7. (Cited on pages 148 and 159.)

[118] Moez Krichen. State Identification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 35–67. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26278-7. DOI: 10.1007/11498490_3. (Cited on page 59.)

[119] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Minimization, Learning, and Conformance Testing of Boolean Programs. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37376-6. DOI: 10.1007/11817949_14. (Cited on pages 119, 135, 141, 145, and 149.)

[120] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Minimization, Learning, and Conformance Testing of Boolean Programs. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2006. URL http://hdl.handle.net/2142/11210. (Cited on pages 141 and 149.)

[121] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly Pushdown Automata for Streaming XML. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1053–1062, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. DOI: 10.1145/1242572.1242714. (Cited on page 144.)

[122] David Lee and Mihalis Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, Mar 1994. ISSN 0018-9340. DOI: 10.1109/12.272431. (Cited on pages 59 and 155.)

[123] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines–a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996. ISSN 0018-9219. DOI: 10.1109/5.533956. (Cited on pages 58 and 68.)

[124] Sylvain Lombardy and Jacques Sakarovitch. The universal automaton. In Wilke Thomas Flum Jörg, Grädel Erich, editor, *Logic and Automata, History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 457–504. Amsterdam University Press, 2008. URL https://hal-upec-upem.archives-ouvertes.fr/hal-00620807. (Cited on pages 59 and 149.)

[125] Nancy A. Lynch and Mark R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM. ISBN 0-89791-239-X. DOI: 10.1145/41840.41852. (Cited on page 151.)

[126] Oded Maler and Irini-Eleftheria Mens. Learning Regular Languages over Large Alphabets. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 485–499. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54861-1. DOI: 10.1007/978-3-642-54862-8_41. (Cited on page 151.)

[127] Oded Maler and Amir Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995. ISSN 0890-5401. DOI: 10.1006/inco.1995.1070. (Cited on pages 49, 51, 58, 110, 148, 150, and 181.)

[128] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. ISBN 0-387-94459-1. (Cited on pages 4, 15, and 16.)

[129] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the Ninth IEEE International High-Level Design Validation and Test Workshop*, pages 95–100, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8714-7. DOI: 10.1109/HLDVT.2004.1431246. (Cited on pages 52, 53, 151, and 152.)

[130] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, 2005. ISSN 1614-5046. DOI: 10.1007/s11334-005-0016-y. (Cited on pages 1, 2, 52, 109, 152, and 153.)

[131] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation. In *Proceedings of the 2005 IEEE International Test Conference (ITC 2005)*. IEEE Computer Society, November 2005. DOI: 10.1109/TEST.2005.1584006. (Cited on pages 152 and 153.)

[132] Karl Meinke. CGE: A Sequential Learning Algorithm for Mealy Automata. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, volume 6339 of *Lecture Notes in Computer Science*, pages 148–162. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15487-4. DOI: 10.1007/978-3-642-15488-1_13. (Cited on pages 46, 59, and 149.)

[133] Karl Meinke and Fei Niu. An Incremental Learning Algorithm for Extended Mealy Automata. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*, ISoLA'12, pages 488–504, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34025-3. DOI: 10.1007/978-3-642-34026-0_36. (Cited on page 149.)

[134] Karl Meinke and Muddassar Azam Sindhu. LBTest: A Learning-Based Testing Tool for Reactive Systems. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013*, pages 447–454, Mar 2013. DOI: 10.1109/ICST.2013.62. (Cited on pages 59 and 85.)

[135] Karl Meinke and John V. Tucker. Universal Algebra. In S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 1)*, pages 189–368. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0-19-853735-2. (Cited on page 149.)

[136] Karl Meinke, F. Niu, and M. Sindhu. Learning-Based Software Testing: A Tutorial. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 200–219. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34780-1. DOI: 10.1007/978-3-642-34781-8_16. (Cited on pages 59, 85, and 149.)

[137] Irini-Eleftheria Mens and Oded Maler. Learning Regular Languages over Large Ordered Alphabets. *CoRR*, abs/1506.00482, 2015. URL http://arxiv.org/abs/1506.00482. (Cited on page 151.)

[138] Maik Merten. *Active automata learning for real-life applications*. PhD thesis, TU Dortmund University, 2013. URL http://hdl.handle.net/2003/29884. (Cited on pages 3, 48, and 148.)

[139] Maik Merten, Falk Howar, Bernhard Steffen, Sofia Cassel, and Bengt Jonsson. Demonstrating Learning of Register Automata. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 466–471. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28755-8. DOI: 10.1007/978-3-642-28756-5_32. (Cited on page 159.)

[140] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata Learning with On-the-Fly Direct Hypothesis Construction. In Reiner Hähnle, Jens Knoop, Tiziana

Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 248–260. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-34780-1. DOI: 10.1007/978-3-642-34781-8_19. (Cited on pages 48 and 148.)

[141] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072. (Cited on page 150.)

[142] Edward F. Moore. Gedanken-Experiments on Sequential Machines. *Annals of Mathematical Studies*, 34:129–153, 1956. (Cited on page 52.)

[143] Daniel Neider and Christof Löding. Learning Visibly One-Counter Automata in Polynomial Time. Technical report, Department of Computer Science, RWTH Aachen, January 2010. URL http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-02.pdf. (Cited on page 149.)

[144] Anil Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939. (Cited on pages 23 and 24.)

[145] Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. Risk-Based Testing via Active Continuous Quality Control. *International Journal on Software Tools for Technology Transfer*, 16(5):569–591, 2014. DOI: 10.1007/s10009-014-0321-6. (Cited on page 153.)

[146] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, Germany, 2003. URL http://hdl.handle.net/2003/2545. (Cited on pages 52, 53, 151, and 152.)

[147] José Oncina and Pedro García. Identifying Regular Languages in Polynomial Time. In *Advance in Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992. (Cited on pages 150 and 153.)

[148] José Oncina, Pedro García, and Enrique Vidal. Learning Subsequential Transducers for Pattern Recognition Interpretation Tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, May 1993. ISSN 0162-8828. DOI: 10.1109/34.211465. (Cited on page 150.)

[149] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black Box Checking. In Jianping Wu, SamuelT. Chanson, and Qiang Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems*, volume 28 of *IFIP Advances in Information and Communication Technology*, pages 225–240. Springer US, 1999. ISBN 978-1-4757-5270-0. DOI: 10.1007/978-0-387-35578-8_13. (Cited on pages 1, 85, and 151.)

[150] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black Box Checking. *J. Autom. Lang. Comb.*, 7(2):225–246, November 2001. ISSN 1430-189X. URL http://dl.acm.org/citation.cfm?id=767345.767349. (Cited on pages 1, 85, and 151.)

[151] Amir Pnueli. In Transition From Global to Modular Temporal Reasoning about Programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer Berlin Heidelberg, 1985. ISBN 978-3-642-82455-5. DOI: 10.1007/978-3-642-82453-1_5. (Cited on page 145.)

[152] Harald Raffelt and Bernhard Steffen. LearnLib: A Library for Automata Learning and Experimentation. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-33093-6. DOI: 10.1007/11693017_28. (Cited on pages 152 and 153.)

[153] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009. ISSN 1433-2779. DOI: 10.1007/s10009-009-0111-8. (Cited on pages 152 and 153.)

[154] Ronald L. Rivest and Robert E. Schapire. Inference of Finite Automata Using Homing Sequences. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 411–420. MIT Laboratory for Computer Science, ACM Press, May 1989. DOI: 10.1145/73007.73047. (Cited on pages 147, 148, and 181.)

[155] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993. ISSN 0890-5401. DOI: 10.1006/inco.1993.1021. (Cited on pages 1, 2, 43, 44, 50, 51, 57, 58, 76, 77, 79, 87, 110, 147, 148, 155, 158, and 181.)

[156] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In Martin Leucker, editor, *Runtime Verification*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89246-5. DOI: 10.1007/978-3-540-89247-2_4. (Cited on page 144.)

[157] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992. ISSN 0890-5401. DOI: 10.1016/0890-5401(92)90003-X. (Cited on page 149.)

[158] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, New York, NY, USA, 2009. ISBN 0521844258, 9780521844253. (Cited on page 16.)

[159] Thomas Schwentick. Automata for XML—A Survey. *Journal of Computer and System Sciences*, 73(3):289–315, 2007. ISSN 0022-0000. DOI: 10.1016/j.jcss.2006.10.003. Special Issue: Database Theory 2004. (Cited on page 144.)

[160] Burr Settles. Active Learning Literature Survey. Technical report, University of Wisconsin, Madison, 2010. URL http://burrsettles.com/pub/settles.activelearning.pdf. (Cited on page 4.)

[161] Muzammil Shahbaz and Roland Groz. Inferring Mealy Machines. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer Verlag. ISBN 978-3-642-05088-6. DOI: 10.1007/978-3-642-05089-3_14. (Cited on pages 49, 51, 52, 53, 58, 110, 151, 152, and 181.)

[162] Guoqiang Shu and David Lee. Testing Security Properties of Protocol Implementations - a Machine Learning Based Approach. In *27th International Conference on Distributed Com-*

puting Systems (ICDCS '07), pages 25–25, June 2007. DOI: 10.1109/ICDCS.2007.147. (Cited on page 53.)

[163] Muddassar Azam Sindhu. *Algorithms and Tools for Learning-based Testing of Reactive Systems*. PhD thesis, School of Computer Science and Communication, KTH Royal Institute of Technology, 2013. URL http://kth.diva-portal.org/smash/get/diva2:610371/FULLTEXT02.pdf. (Cited on page 85.)

[164] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. Applying Active Automata Learning to Embedded Control Software. In *Proceedings of the* 17^{th} *International Conference on Formal Engineering Methods (ICFEM 2015)*, September 2015. to appear. (Cited on page 59.)

[165] Rick Smetsers and Joshua Moerman. Minimal Separating Sequences for All Pairs of States. Technical report, Institute for Computing and Information Sciences, Radboud University Nijmegen, 2005. URL http://cs.ru.nl/~rick/files/sm2015.pdf. (Cited on page 68.)

[166] Jiří Srba. Visibly Pushdown Automata: From Language Equivalence to Simulation and Bisimulation. In Zoltán Ésik, editor, *Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-45458-8. DOI: 10.1007/11874683_6. (Cited on page 118.)

[167] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-21455-4_8. (Cited on pages 3, 31, 52, 58, and 155.)

[168] Boris A. Trakhtenbrot and Ya M. Barzdin. *Finite Automata: Behavior and Synthesis*. American Elsevier Publishing Company, 1973. (Cited on page 1.)

[169] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973. ISSN 0011-4235. DOI: 10.1007/BF01068590. (Cited on page 152.)

[170] Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris. Increasing functional coverage by inductive testing: A case study. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS'10, pages 126–141, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16572-9, 978-3-642-16572-6. DOI: 10.1007/978-3-642-16573-3_10. (Cited on page 150.)

[171] Stephan Windmüller. *Kontinuierliche Qualitätskontrolle von Webanwendungen auf Basis maschinengelernter Modelle*. Dissertation, Technische Universität Dortmund, July 2014. URL http://hdl.handle.net/2003/33540. (Cited on page 153.)

[172] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active Continuous Quality Control. In *16th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE '13, pages 111–120. ACM SIGSOFT, New York, NY, USA, 2013. DOI: 10.1145/2465449.2465469. (Cited on page 153.)

[173] Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. TzuYu: Learning stateful typestates. In *28th International IEEE/ACM Conference on Automated Software Engineering (ASE 2013)*, pages 432–442, Nov 2013. DOI: 10.1109/ASE.2013.6693101. (Cited on page 1.)

[174] Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The State Complexities of Some Basic Operations on Regular Languages. *Theor. Comput. Sci.*, 125(2):315–328, March 1994. ISSN 0304-3975. DOI: 10.1016/0304-3975(92)00011-F. (Cited on page 22.)

[175] Sandra Zilles, Steffen Lange, Robert Holte, and Martin Zinkevich. Models of Cooperative Teaching and Learning. *J. Mach. Learn. Res.*, 12:349–384, February 2011. ISSN 1532-4435. URL http://www.jmlr.org/papers/volume12/zilles11a/zilles11a.pdf. (Cited on pages 26, 87, and 135.)

A. Supplementary Material

A.1. Overview of Active Learning Algorithms' Complexities

Name	Ref.	Query Complexity	Symbol Complexity
L*	[19]	$\mathcal{O}(kn^2m)$	$\mathcal{O}(kn^2m^2)$
L^*_{col}	[127]	$\mathcal{O}(kn^2m)$	$\mathcal{O}(kn^2m^2)$
Shahbaz	[106, 161]	$\mathcal{O}(kn^2m)$	$\mathcal{O}(kn^2m^2)$
Suffix1by1	[105, 106]	$\mathcal{O}(kn^2m)$	$\mathcal{O}(kn^2m^2)$
Rivest/Schapire	[154, 155]	$\mathcal{O}(kn^2+n\log m)$	$\mathcal{O}(kn^2m+nm\log m)$
Kearns/Vazirani (orig.)	[115]	$\mathcal{O}(kn^2+n^2m)$	$\mathcal{O}(kn^2m+n^2m^2)$
Kearns/Vazirani (bin. search.)	[108]	$\mathcal{O}(kn^2+n^2\log m)$	$\mathcal{O}(kn^2m+n^2m\log m)$
Observation Pack	[93, 108], Sec. 4.2.3	$\mathcal{O}(kn^2+n\log m)$	$\mathcal{O}(kn^2m+nm\log m)$
TTT	[110], Chp. 5	$\mathcal{O}(kn^2+n\log m)$	$\mathcal{O}(kn^2m+nm\log m)$

Table A.1.: Query and symbol complexities of active automata learning algorithms