Applications of Automata Learning in Verification and Synthesis

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker, Diplom-Wirtschaftsinformatiker
Hermann Daniel Neider
aus Remscheid

Berichter: Universitätsprofessor Dr. Dr. h. c. Dr. h. c. Wolfgang Thomas
Privatdozent Dr. Christof Löding
Universitätsprofessor Dr. Martin Leucker

Tag der mündlichen Prüfung: 25. April 2014

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Abstract

The objective of this thesis is to explore automata learning, which is an umbrella term for techniques that derive finite automata from external information sources, in the areas of verification and synthesis. We consider four application scenarios that turn out to be particularly well-suited: Regular Model Checking, quantified invariants of linear data structures, automatic reachability games, and labeled safety games. The former two scenarios stem from the area of verification whereas the latter two stem from the area of synthesis (more precisely, from the area of infinite-duration two-player games over graphs, as popularized by McNaughton).

Regular Model Checking is a special kind of Model Checking in which the program to verify is modeled in terms of finite automata. We develop various (semi-)algorithms for computing invariants in Regular Model Checking: a white-box algorithm, which takes the whole program as input; two semi-black-box algorithms, which have access to a part of the program and learn missing information from a teacher; finally, two black-box algorithms, which obtain all information about the program from a teacher. For the black-box algorithms, we employ a novel paradigm, called ICE-learning, which is a generic learning setting for learning invariants.

Quantified invariants of linear data structures occur in Floyd-Hoare-style verification of programs manipulating arrays and lists. To learn such invariants, we introduce the notion of quantified data automata and develop an active learning algorithm for these automata. Based on a finite sample of configurations that manifest on executions of the program in question, we learn a quantified data automaton and translate it into a logic formula expressing the invariant. The latter potentially requires an additional abstraction step to ensure that the resulting formula falls into a decidable logic.

Automatic reachability games are classical reachability games played over automatic graphs; automatic graphs are defined by means of asynchronous transducers and subsume various types of graphs, such as finite graphs, pushdown graphs, and configuration graphs of Turing machines. We first consider automatic reachability games over finite graphs and present a symbolic fixed-point algorithm for computing attractors that uses deterministic finite automata to represent sets of vertices. Since such a

fixed-point algorithm is not guaranteed to terminate on games over infinite graphs, we subsequently develop a learning-based (semi-)algorithm that learns the attractor (if possible) from a teacher rather than computing it iteratively.

Finally, we consider *labeled safety games*, which are safety games played over finite graphs whose edges are deterministically labeled with actions. The problem we are interested in is to compute a winning strategy that, when implemented as as a circuit or a piece of code, results in a “small” implementation. To this end, we model strategies as so-called strategy automata and exploit a common property of active learning algorithms, namely that such algorithms produce conjectures of increasing size. The idea is to start learning the set of all winning plays and stop the learning prematurely as soon as the learner conjectures a winning strategy automaton. This procedure guarantees that the resulting strategy automaton is at most as large as the underlying game graph. To improve the performance of our algorithm further, we develop domain-specific optimizations of Angluin’s as well as Kearns and Vazirani’s active learning algorithms.

For my parents,
who were a great example for me.

Acknowledgement

Over the past five years, I have received support and encouragement from a great number of individuals, who have contributed to this thesis both directly and indirectly.

First and foremost, I want to thank my advisor Christof Löding, who brought automata learning to my attention and encouraged me to work on this topic. His guidance has made this a thoughtful and rewarding journey.

I am also very grateful to Wolfgang Thomas for giving me the opportunity to work with him and write this dissertation. He constantly supported me as I tried to juggle teaching and conducting my research.

Furthermore, I would like to thank Martin Leucker and Wolfgang Thomas for co-examining my thesis and Joost-Pieter Katoen and Thomas Seidl for completing my thesis committee.

Many of the results presented here were obtained in collaborations: I want to thank my co-authors Benedikt Bollig, Pranav Garg, Nils Jansen, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Christof Löding, P. Madhusudan, David Piegdon, Roman Rabinovich, and Martin Zimmermann. I am also grateful to P. Madhusudan for inviting me to visit him.

I enjoyed the time with my colleagues very much. I am grateful for interesting discussions (mainly during the coffee breaks), many insights into all sorts of topics, and a lot of fun.

Finally, I want to thank my family, in particular my parents and my girlfriend Nicole, for their constant support. They were always there to pick me up when things went wrong.

Contents

1 *Introduction* ♦ *1*

Verification ♦ 3

Synthesis ♦ 7

2 *Preliminaries* ♦ *11*

2.1 Finite Automata, Transducers, and Rational Graphs ♦ 12

2.2 First-order Logics ♦ 19

2.3 Infinite Games ♦ 34

3 *Algorithmic Learning of Finite Automata* ♦ *41*

3.1 Passive Learning of DFAs ♦ 42

3.1.1 Biermann and Feldman's Method ♦ 45

3.1.2 Grinchtein, Leucker, and Piterman's Unary SAT Method ♦ 48

3.1.3 Grinchtein, Leucker, and Piterman's Binary SAT Method ♦ 50

3.1.4 Heule and Verwer's SAT Method ♦ 53

3.1.5 An SMT-based Method ♦ 55

3.1.6 Evaluation ♦ 57

3.2 Active Learning of DFAs ♦ 62

3.2.1 Angluin's Learning Algorithm ♦ 63

3.2.2 Kearns and Vazirani's Learning Algorithm ♦ 71

3.2.3 Comparison ♦ 77

3.3 Libalf: the Automata Learning Framework ♦ 78

4 *Regular Model Checking* ♦ *81*

Related Work ♦ 87

4.1 Regular Model Checking and Invariants ♦ 90

4.2 A White-box Algorithm ♦ 93

4.2.1 Computing IDFAs Using Propositional Boolean Logic ♦ 95

4.2.2 Computing IDFAs Using the Logic of Uninterpreted Functions over the Natural Numbers ♦ 99
4.3 Semi-black-box Algorithms ♦ 102
4.3.1 An Incomplete Teacher for Regular Model Checking ♦ 104
4.3.2 Passive Learning of Inductive DFAs from Samples and Transducers ♦ 104
4.3.3 The CEGAR-style Learner ♦ 108
4.3.4 The Angluin-style Learner ♦ 110
4.4 Black-box Algorithms ♦ 113
4.4.1 An ICE-teacher for Regular Model Checking ♦ 115
4.4.2 Passive Learning of DFAs from Samples and Implications ♦ 116
4.4.3 CEGAR-Style ICE-learner for Regular Model Checking ♦ 120
4.4.4 Angluin-Style ICE-learner for Regular Model Checking ♦ 120
4.5 Experiments and Evaluation ♦ 122
4.6 Further Applications ♦ 128
4.6.1 Computing Minimal Separating DFAs ♦ 129
4.6.2 Synthesizing Presburger Loop Invariants ♦ 130
4.7 Conclusion ♦ 133

5 ***Quantified Invariants of Linear Data Structures*** ♦ *135*

Related Work ♦ 141
5.1 Quantified Data Automata ♦ 142
5.2 Learning Quantified Data Automata ♦ 147
5.2.1 Canonical QDAs ♦ 147
5.2.2 Learning QDAs by Learning Moore Machines ♦ 149
5.3 Elastic Quantified Data Automata ♦ 152
5.4 Converting Program Configurations to Data Words and EQDAs to Logics ♦ 155
5.4.1 Modeling Program Configurations as Data Words ♦ 155
5.4.2 Converting EQDAs to Strand and the Array Property Fragment ♦ 157
5.5 Learning Universally Quantified Invariants ♦ 172
5.6 Experiments and Evaluation ♦ 174
5.7 Conclusion ♦ 179

6 ***Automatic Reachability Games*** ♦ *181*

Related Work ♦ 183
6.1 Automatic Arenas and Automatic Reachability Games ♦ 183
6.2 Automatic Reachability Games over Finite Arenas ♦ 185
6.3 Automatic Reachability Games over Infinite Arenas ♦ 188
6.3.1 An Alternative Characterization of the Attractor ♦ 188

6.3.2 A Teacher for Automatic Reachability Games ♦ 191
6.3.3 Solving Automatic Reachability Games over Infinite Arenas ♦ 193
6.4 Experiments and Evaluation ♦ 197
6.4.1 A Framework to Evaluate Symbolic Techniques for Reachability Games ♦ 197
6.4.2 Experiments ♦ 200
6.5 Conclusion ♦ 204

7 *Labeled Safety Games* ♦ *207*

Related Work ♦ 209
7.1 Labeled Safety Games and Strategy Automata ♦ 210
7.1.1 Labeled Safety Games ♦ 210
7.1.2 Implementation of Strategies and Strategy Automata ♦ 213
7.1.3 Size of Strategy Implementations ♦ 216
7.1.4 Minimal Strategy Automata ♦ 218
7.2 Learning Small Strategy Automata ♦ 223
7.2.1 A Teacher for Strategy Automata ♦ 224
7.2.2 Two Domain-specific Learning Algorithms ♦ 226
7.2.3 Results of Learning Strategy Automata ♦ 235
7.3 Experiments and Evaluation ♦ 236
7.4 Conclusion ♦ 240

8 *Conclusion* ♦ *243*

Bibliography ♦ *249*

Index ♦ *265*

List of Figures

2.1 A DFA accepting the language $L = \{u \in \{a,b\}^* \mid |u| \text{ is odd}\}$ ♦ 14

2.2 An asynchronous transducer that defines the relation $\{(ab^n, ab^{n+1}) \mid n \geq 0\} \cup \{(ab^{n+2}, ab^n) \mid n \geq 0\}$ ♦ 16

2.3 The rational graph $G = (V, E)$ with $V = ab^*$ and $E = R(\mathcal{T})$ where $\mathcal{T}$ is the asynchronous transducer of Figure 2.2 ♦ 19

2.4 The DFA $\mathcal{A}^{\varphi}$ accepting the set of words that encode integers satisfying the formula $\varphi(x,y,z) \coloneqq x + y = z$ ♦ 27

2.5 The game $\mathfrak{G}^{\star}$ discussed in Example 2.10 ♦ 37

3.1 The prefix acceptor $\mathcal{A}^{\mathcal{S}}_{\text{Pref}}$ for the sample $\mathcal{S} = (S_+, S_-)$ where $S_+ = \{aa, ba, aba\}$ and $S_- = \{\varepsilon, ab\}$ and a minimal consistent DFA ♦ 44

3.2 Performance of the passive learning algorithms described in Section 3.1 ♦ 61

3.3 The sequence of observation tables produced during the run of Angluin's algorithm in Example 3.2 ♦ 68

3.4 Classification trees and conjectures produced in Example 3.3 ♦ 74

4.1 Automata modeling the correctness of the token ring protocol as a Regular Model Checking problem ♦ 92

4.2 An IDFA for the Regular Model Checking problem of Example 4.1 ♦ 93

4.3 A While program and Presburger formulas annotating the While loop ♦ 130

5.1 A program configuration of a fictitious program manipulating a list and the corresponding valuation word ♦ 137

5.2 A QDA accepting the set of data words that correspond to program configurations satisfying Formula 5.1 ♦ 138

5.3 A valuation word together with a depiction of which components constitute the corresponding data word and symbolic word ♦ 144

5.4 A QDA expressing that the data on even list cells is sorted ♦ 148

5.5 Base cases of the inductive definition of irrelevant self-loops ♦ 158
5.6 A simple path of an EQDA that was learned in our experiments ♦ 164
5.7 Two diverging simple paths ♦ 167

6.1 The reachability game of Example 6.1 ♦ 184
6.2 An automatic representation of the reachability game of Example 6.1 ♦ 185
6.3 The pursuit-evasion game ♦ 198
6.4 The swap game ♦ 198

7.1 An example of a labeled safety game over an $\{a, b\}$-arena ♦ 211
7.2 A strategy automaton that is winning for Player 0 in the labeled safety game of Figure 7.1 from vertex v_0 ♦ 215
7.3 A family $(\mathfrak{G}_n)_{n \in \mathbb{N}}$ of safety games and a winning strategy automaton ♦ 217
7.4 The safety game $\mathfrak{G}_\varphi$ and a winning strategy automaton ♦ 219
7.5 A safety game and a corresponding winning strategy automaton learned by our prototype ♦ 237
7.6 Experimental results of our prototype on random safety games ♦ 240

List of Tables

3.1 An overview of popular learning algorithms ♦ 41
3.2 Overview of the passive learning algorithms described in Section 3.1 ♦ 58
3.3 An overview of popular active learning algorithms ♦ 77
3.4 Summary of learning algorithms implemented in Libalf ♦ 79

4.1 Feature summary of algorithms for Regular Model Checking ♦ 124
4.2 Experimental results on integer linear systems ♦ 126
4.3 Experimental results on modulo counter experiments ♦ 127
4.4 Experimental results on token ring experiments ♦ 127
4.5 Experimental results of our prototype implementation on a selection of InvGen's "C test suite" ♦ 133

5.1 Experimental results on learning loop invariants ♦ 177
5.2 Experimental results on learning function preconditions ♦ 177

6.1 Experimental results of the DFA-based attractor computation on games over finite arenas ♦ 201
6.3 Experimental results of the learning-based algorithm on the unbounded swap game ♦ 202
6.4 Summary of the benchmark by Alur, Madhusudan, and Nam and our experimental results on reachability games over finite arenas ♦ 203

List of Algorithms

3.1 Angluin's active learning algorithm ♦ 67
3.2 Kearns and Vazirani's active learning algorithm ♦ 73

4.1 Computing IDFAs using logic solver ♦ 94
4.2 Passive learning algorithm for learning inductive DFAs ♦ 105
4.3 CEGAR-style learner for Regular Model Checking ♦ 109
4.4 Angluin-style learner for Regular Model Checking ♦ 112
4.5 Passive learning algorithm for learning DFAs that respect implications ♦ 116
4.6 CEGAR-style ICE-learner for Regular Model Checking ♦ 121
4.7 Angluin-style ICE-learner for Regular Model Checking ♦ 122

6.1 DFA-based symbolic attractor computation ♦ 187

7.1 Learning-based algorithm for synthesizing strategy automata ♦ 224
7.2 A variant of Angluin's active learning algorithm ♦ 229
7.3 A variant of Kearns and Vazirani's active learning algorithm ♦ 233

„Ein Jahrhundert, das sich bloß auf die Analyse verlegt, und sich vor der Synthese gleichsam fürchtet, ist nicht auf dem rechten Wege; denn nur beide zusammen, wie Aus- und Einatmen, machen das Leben der Wissenschaft.“

Johann Wolfgang von Goethe

1 Introduction

When Angluin introduced her automata learning framework in 1987, the topic seemed to be of mere theoretical interest, and practical applications did not emerge for a long time. Since then, this has changed completely.

During the last decade, the study of automata learning has gained significant momentum. Many successful applications were reported, ranging from pattern and natural language recognition over computational biology, data mining, and robotics to automatic verification and even the analysis of music (see de la Higuera [dlH05] for an extensive survey). In short, automata learning is of broad and current interest.

Automata learning comes in many flavors, and one usually distinguishes two major categories: passive learning, often attributed to Biermann and Feldman [BF72] as well as Trakhtenbrot and Barzdin [TB73], and active learning due to Angluin [Ang87].

Passive learning In the passive learning setting, a learning algorithm—commonly abbreviated as *learner*—is confronted with a finite set of classified words, and the task is to find some (often a smallest) finite automaton that classifies the given words correctly. A popular setting is one in which the words are classified either positively or negatively, and the learner has to find a smallest deterministic finite automaton that accepts all positively classified words and rejects all negatively classified ones. Sometimes, passive learning is used iteratively to refine the automata that have been learned in previous iterations whenever new information becomes available. This idea is similar to the *counterexample guided abstraction refinement (CEGAR)* principle [CGJ+00], a popular technique in Model Checking.

Active learning In the active learning setting, a learner learns a language, called *target language*, in interaction with an information source, which is usually termed

teacher. Active learning is similar to the human learning process and proceeds in rounds: the learner successively poses queries, which the teacher answers, until the learner has eventually learned the concept the teacher has in mind. In Angluin's original setting, for instance, the learner learns a regular language from a teacher by means of *membership* and *equivalence queries*: the former is the question of whether a word belongs to the target language, whereas the latter is the question of whether a conjectured automaton recognizes the target language. If an automaton fails an equivalence query, the teacher is required to return a *counterexample* (i.e., a word that witnesses a difference between the conjectured automaton and the target language).

Many learning algorithms for both the passive and the active learning setting have been developed. Among the best-known is the *regular positive negative inference (RPNI)* passive learning algorithm due to Oncina and Garcia [OG92] as well as Angluin's active learning algorithm [Ang87].

The popularity of automata learning arises from a number of beneficial properties that these algorithms enjoy. Among the most significant are the following:

- Automata learning algorithms typically identify a minimal automaton (or at least a "small" automaton) that agrees with the available information. The motivation for this is *Occam's razor*, namely that a simple (i.e., small) automaton is the best explanation of the given information.

- Automata learning algorithms are good at generalizing knowledge. By this, we refer to the fact that automata learning algorithms construct automata, usually accepting infinite sets of words, from a finite amount of information.

- Automata learning naturally separates the source of information from the actual learning algorithm and provides a standard protocol that defines how information is to be exchanged. This is especially true in the active learning setting, where both the teacher and the learner can be optimized for their respective task: the learner tries to generalize as much as possible from the information learned so far and strives for a small automaton, whereas the teacher focuses on answering queries, which he can often do by just considering a small fragment of the (potentially complex) target language.

Clearly, automata learning also has drawbacks. For instance, active learning has the issue that the runtime of a learning algorithm depends on the "quality" of counterexamples; if the teacher replies an unnecessary long counterexample, the learner has no choice but to process the whole word in order to make progress. A similar situation can occur in passive learning if the given sample contains much redundant

information. However, the positive aspects of automata learning often outweigh the negative, which makes automata learning a valuable tool for many practical problems.

Motivated by the success of automata learning techniques, the objective of this thesis is to lift automata learning to the areas of *(formal) verification* and *synthesis*. We explore four application scenarios that turn out to be particularly well-suited: Regular Model Checking, quantified invariants of linear data structures, automatic reachability games, and labeled safety games. The first two scenarios stem from the area of formal verification whereas the latter two stem from the area of synthesis.

The remainder of this chapter gives the gist of these four scenarios—while detailed introductions follow in the main part (in Chapters 4 to 7)—and outlines the contributions of this thesis. So as not to clutter this introduction too much, putting our work into scientific context (in particular, citing and discussing related work) is done per topic in the corresponding chapters. This introduction concludes with information about how this thesis is intended to be read.

Verification

Despite advances in software engineering, software development remains an error-prone activity, and ensuring reliability is an integral part of the development process. An approach of increasing importance in this context is (formal) *verification*, which is the task to check automatically whether the program in question satisfies or violates certain user-specified properties. D'Silva, Kroening, and Weissenbacher [DKW08] survey numerous general approaches to software verification, and the survey by Leucker [Leu06] gives an excellent overview of automata learning in this context.

In this thesis, we apply automata learning to two popular verification approaches, namely Model Checking [CGP01] and Floyd-Hoare-style reasoning [Flo67, Hoa69]. First, we consider a special case of Model Checking called *Regular Model Checking*; in Regular Model Checking, the program is given in terms of regular languages, which makes it a natural choice for the application of automata learning since the problem is already phrased in a suitable format. Second, we consider the problem of computing quantified invariants of linear data structures, which are used for Floyd-Hoare-style verification of programs manipulating arrays or lists; in this context, we exploit that linear data structures can be seen as words, which allows applying automata learning.

Regular Model Checking

Regular Model Checking [KMM+97, BJNT00] is a general approach to verifying infinite-state systems, which arise naturally if a program has access to queues or stacks, manipulates data of an infinite domain, and so on. In Regular Model Checking,

configurations of a program are modeled as finite words, sets of configurations as regular languages, and the program's semantics as a *rational* relation (defined by an asynchronous transducer). Regular Model Checking is typically used to verify safety properties—although it is not restricted to properties of this kind—and was successfully applied to various real-world problems [Leg08, BFL04, AJNS04]. However, since the Regular Model Checking problem is undecidable in general, algorithms for Regular Model Checking are necessarily incomplete.

Existing tools for Regular Model Checking such as T(O)RMC [Leg08] operate by manipulating the given input-automata in order to compute a finite automaton that accepts an invariant of the program at hand; an invariant is a set of program configurations that contains all initial configurations, no configuration that has been labeled as bad, and is closed under the program's transition relation. Although this is an effective approach, it has drawbacks. On the one hand, it requires the program to be expressible in terms of (deterministic) finite automata. On the other hand, manipulating the input-automata directly often produces large intermediate results and seriously limits the algorithm's applicability.

Learning based algorithms have also been proposed. One example is the tool LEVER, which actively learns an invariant via an elaborate, intermediate fixed point. Despite the fact that the authors report good results on examples, the tool is no longer maintained and not publicly available. Another algorithm was proposed by Habermehl and Vojnar [HV05]. Their idea is to sample all words of length smaller or equal to a value i with respect to the program at hand and apply the Trakhtenbrot-Barzdin passive learning algorithm [TB73]; if no valid invariant has been found this way, the parameter i is increased and the procedure is repeated. Although Habermehl and Vojnar report good performance results on selected experiments, their approach has clearly the disadvantage that it samples an exponential number of words, which seriously limits its applicability in practical instances.

Contributions We develop several algorithms for Regular Model Checking that avoid the disadvantages mentioned above. Our algorithms share the pivotal idea of using state-of-the-art logic solvers as a viable and established means to handle expensive calculations.

We begin with an algorithm that is not yet based on learning but serves as the foundation of our learning-based algorithms. The idea is to search for an invariant in form of a deterministic finite automaton of a fixed size n. We postulate the existence of such an automaton in terms of a logic formula φ_n and use a logic solver to solve the formula; if the formula is unsatisfiable, n is increased until φ_n becomes satisfiable (which is guaranteed to happen if and only if such an automaton exists). Additionally, φ_n is designed such that a model provides all information necessary to construct

a deterministic finite automaton that accepts an invariant. We consider two logics to draw φ_n from, namely Propositional Boolean Logic and a logic of uninterpreted functions over the natural numbers.

Building upon this algorithm, we develop a number of algorithms combining active learning and state-of-the-art logic solvers, which aim at learning an invariant of the program rather than computing one explicitly. Following the general idea of active learning, our algorithms divide this task into two parts: the teacher focuses on reasoning about the (potentially complex) program, whereas the learner, unaware of the program's complexity, learns a (smallest) invariant by means of a logic solver (if one exists). This combination of active learning and solver technologies allows us to separate reasoning about the program from the actual invariant synthesis and to delegate costly computations to highly optimized logic solvers. Our learning-based algorithms can be grouped into two categories: algorithms of the first category operate according to the CEGAR principle, whereas those of the second category extend the ideas of Angluin's algorithm.

The greatest challenge of our approach stems from the fact that the teacher does not know an invariant in advance. So as to still use automata learning, we introduce a novel learning setting called *ICE-learning* (meaning learning from implications, counterexamples, and examples); in this setting, our teacher can reply implications of the form $u \rightarrow v$, meaning that v has to be accepted if u is accepted. This allows the teacher to answer queries despite the fact that he does not know an invariant beforehand.

Quantified Invariants of Linear Data Structures

Floyd-Hoare-style verification crucially relies upon loop invariants. However, providing loop invariants is a difficult and burdensome task, and much research has been spent on mechanisms that automatically synthesize loop invariants in order to reduce the manual work otherwise necessary.

Since linear data structures are a common feature in today's software, we focus on programs that use arrays and lists. An important role in this context play universally quantified invariants of the form

$$\forall y_1 \colon \ldots \forall y_k \colon \varphi(y_1, \ldots, y_k),$$

where the universally quantified variables $y_1, \ldots, y_k$ reference array entries, respectively list cells, and φ is a quantifier-free formula relating the data referenced by the universally quantified variables. In fact, such invariants are sufficient to express many properties, such as sortedness, uniqueness of entries, and so on.

Our goal is to develop a learning-based algorithm for learning quantified invariants of linear data structures. As in the case of Regular Model Checking, our motivation for using automata learning is the ability to split the problem into two parts: a teacher, who reasons about the (potentially complex) program, and a learner, who focuses on synthesizing an invariant.

Contributions To capture quantified invariants of linear data structures, we model linear data structures as finite words over a potentially infinite alphabet, called *data words*, and introduce a novel automaton model called *quantified data automata*, which operates on data words. Quantified data automata are equipped with formulas at every state and accept a data word if and only if the formula decorating the state finally reached is satisfied for all possible valuations of the universally quantified variables. In this way, a quantified data automaton uniquely corresponds to a set of data words and, hence, to a set of linear data structures.

Building upon quantified data automata, we develop a framework for learning quantified invariants of linear data structures. This framework consists of the following components:

- An Angluin-style active learning algorithm for quantified data automata
- A translation of a subclass of quantified data automata, called *elastic quantified data automata*, into decidable logics over linear data structures (the Array Property Fragment [BMS06] for arrays and the decidable syntactic fragment of Strand [MPQ11] for lists)
- A mechanism to abstract from quantified data automata to elastic quantified data automata

Learning a universally quantified formula now amounts to learning a quantified data automaton, abstract it into an elastic quantified data automaton (if necessary), and translate the result into a logic formula.

In order to reduce the overall complexity of our approach, we avoid working in an active (ICE-)learning setting as this would require to construct a teacher who is able to reason about complex program logic (such as manipulations of data structures or other heap operations). Instead, we follow a light-weight approach in which we apply our active learning algorithm in a passive learning setting. More precisely, we first run the program exhaustively on “short” lists, respectively arrays, populated with data from a “small” finite domain to collect a finite sample of data structures that manifest during the program’s execution. Then, we construct a teacher who answers queries with respect to this sample; whenever the learner queries for information the teacher does not possess, he returns an arbitrary answer. Although this is an

incomplete approach, our experiments show that such a teacher is often sufficient to learn an adequate invariant because invariants typically depend neither on length of the data structures nor on the exact data contained therein.

Synthesis

A complementary approach to making software reliable is *synthesis*, which refers to the task of automatically synthesizing a circuit or a piece of code from specifications. In contrast to verification, synthesis removes the burden of writing code from the developer and guarantees that the resulting implementation is correct with respect to the given specification.

We are interested in a particular kind of synthesis, called *controller synthesis*, which has its roots in the analysis of reactive systems and dates back to the 1950s. Church [Chu57] considered a scenario in which a *system* receives an infinite sequence of input-symbols from an *environment*, symbol by symbol, and replies to each input-symbol with an output-symbol. The system's objective is to ensure for every possible input-sequence of the environment that the infinite sequence of inputs and outputs satisfies an a priori given specification. The corresponding question, namely whether a finite-state controller for the system's task can be constructed automatically, is today known as *Church's synthesis problem*.[1]

A prevalent view on Church's synthesis problem, which has mainly been influenced by McNaughton [McN93], is to understand the problem in terms of infinite two-person games over (finite) graphs, *infinite games* for short. Such games are played by two players, Player 0 and Player 1, who move a token from one vertex to the next along the edges of the graph. The specification is interpreted as a winning condition that expresses conditions on sequences of vertices, and the objective of Player 0 is to fulfill the winning condition regardless of the moves of Player 1. From this point of view, Church's synthesis problem amounts to deciding whether Player 0 can win the infinite game, and a (finite-state) winning strategy corresponds to a (finite) implementation of a controller that satisfies the given specification.

To the best of our knowledge, automata learning has not yet been explored in the context of infinite games. As a first step towards bridging this gap, we apply automata learning to two fundamental types of infinite games: first, we consider *reachability games*, in which Player 0's objective is to reach a designated set of vertices; second, we consider *safety games*, which are dual to reachability games in that Player 0 has to avoid a designated set of vertices.

[1] A series of papers by Thomas [Tho08a, Tho08b, Tho09] investigates Church's synthesis problem from various perspectives.

Reachability and safety games form an important subclass of infinite games because they result from guarantee (respectively safety) specifications, which are a fundamental class of specifications occurring in software engineering (Dwyer, Avrunin, and Corbett [DAC99] give a comprehensive overview of typical specification used in practice). Additionally, we recently showed that games with more complex winning conditions (i.e., Muller and, consequently, parity and Büchi winning conditions) can be "reduced" to safety games in order to compute winning regions and winning strategies [NRZ12]. Thus, safety games serve as a "normal form" for infinite games with ω-regular winning conditions.

Automatic Reachability Games

Infinite games, and reachability games in particular, are usually studied over finite graphs (see Grädel, Thomas, and Wilke [GTW02] for an overview). However, infinite graphs arise naturally as soon as one of the players has access to an unbounded data structure or works with data over an unbounded domain.

Dealing with games over infinite graphs entails two major problems. The first is the question of how to represent an infinite graph as a finite object in order to use it as input of an algorithm. The second is that classical algorithms (as used in the case of finite graphs) are no longer guaranteed to terminate, and both determining the winner of a game and computing winning strategies requires more elaborate methods if the game graph is infinite.

Contributions To tackle the first problem, we introduce *automatic reachability games*. Similar to Regular Model Checking, automatic reachability games are defined in terms of regular languages (represented as finite automata); the underlying game graphs, however, are *automatic* rather than rational and form a proper subclass of rational graphs. Nonetheless, automatic graphs subsume a variety of graphs, including finite graphs, pushdown graphs, and even configuration graphs of Turing machines; the latter, of course, implies that deciding reachability properties is not possible in general. As in the case of Regular Model Checking, the formulation in terms of regular language makes automatic reachability games a natural scenario to apply automata learning.

To tackle the second problem, we initially develop a symbolic fixed-point algorithm (not yet based on automata learning) for solving[2] automatic reachability games over finite graphs. Building on that, we introduce an active learning framework for solving automatic reachability games over infinite graphs. The key property of this framework

[2]The term *solving a game* refers to the process of determining the winner and computing winning strategies.

is that it defines a unique target language for any given automatic reachability game from which one can derive both the winner of the game and winning strategies. The uniqueness of the target language makes it possible to construct a teacher who answers queries precisely and to apply off-the-shelf active learning algorithms.

Labeled Safety Games

Much research has been spent on efficient algorithms to solve various types of infinite games. From a software engineering point of view, however, it is often less important how fast a winning strategy can be computed but how much space is actually needed to implement the strategy (e.g., in terms of the size of a circuit or the number of instructions of a piece of software). In fact, this question seems hard to settle, and there is no satisfactory answer until today. Even harder is the task of computing a minimal—or at least "small"—implementation of a winning strategy.

We study the question of how to compute small implementations of strategies in the context of safety games and develop an algorithm based on active learning for this task. A straightforward way to make classical safety games amenable to automata learning is to consider *labeled safety games*: safety games of this kind are played on finite graphs whose edges are deterministically labeled with actions, and the players choose actions rather than successor vertices. Labeled safety games allow us to view strategies as sets of action sequences and to represent (winning) strategies in terms of deterministic finite automata. This view makes it possible to apply automata learning techniques.

However, the problem of finding a deterministic finite automaton of minimal size that represents a winning strategy is computationally hard and cannot even be approximated in polynomial time [Ehl11]. Thus, our goal is a heuristic that aims for "small" but not necessarily minimal automata. To this end, we exploit a common property of active learning algorithms, namely that such algorithms typically produce conjectures of monotonically increasing size.

Contributions As first step, we define (winning) strategies in terms of deterministic finite automata called *(winning) strategy automata*. Strategy automata enjoy two key properties: on the one hand, they provide a natural measure for the size of a strategy implementation, namely the number of states; on the other hand, if a player can win a labeled safety game, then one can efficiently construct a unique *canonical strategy automaton* that implements a winning strategy for this player.

Based on this notion, we then develop a heuristic for learning small winning strategy automata. The underlying idea is the following: for a given labeled safety game, we use the language accepted by the canonical strategy automaton as target language and

construct a corresponding teacher; then, we pit this teacher against an active learning algorithm that produces conjectures of monotonically increasing size and terminate the learning as soon as the learner conjectures a winning strategy automaton (which the teacher can efficiently check). This procedure guarantees that the learned strategy automaton is at most as large as the canonical strategy automaton.

Due to the heuristic nature of this approach, however, we cannot make guarantees about the size of the resulting strategy automaton. In fact, it turns out that standard active learning algorithms often learn the canonical strategy automaton exactly. To avoid this problem, we design modifications of both Angluin's algorithm and Kearns and Vazirani's algorithm. In contrast to the original algorithms, these modifications explore transitions only if a counterexample proves their necessity rather than every time a new state is discovered. When using these modified algorithms, it turns out that the learning almost always terminates early, hence, resulting in a small winning strategy automaton.

Outline

The remainder of this thesis is organized around the four application scenarios. We begin in Chapter 2 by fixing notations and definitions used throughout the thesis. Subsequently, Chapter 3 gives a general introduction to both active and passive automata learning, and presents the learning algorithms used in this thesis in detail. Additionally, this chapter contains a comprehensive experimental comparison of passive learning algorithms and briefly presents the LIBALF automaton learning library.

Chapters 4 to 7 constitute the main part of this thesis. Chapter 4 deals with Regular Model Checking, Chapter 5 with quantified invariants of linear data structures, Chapter 6 with automatic reachability games, and Chapter 7 with labeled safety games. These chapters are self-contained and can be read in any order. In particular, each chapter comprises a separate introduction, a summary of related work, and a conclusion. Moreover, each chapter provides a comprehensive experimental evaluation of the developed algorithms based on prototype implementations; to put our algorithms into context, we contrast these prototypes to existing work (whenever such work exists).

Finally, Chapter 8 concludes with a recapitulation and summarizes open questions as a guide to future research.

2

Preliminaries

This chapter introduces basic definitions and notations that we use throughout this thesis. We start in Section 2.1 by recapping fundamentals of formal language theory, including *words* and *languages*, *finite automata*, and *transducers*; this section also covers *automatic* and *rational graphs*. In Section 2.2, we discuss *first-order logic* and several restrictions thereof. Finally, Section 2.3 covers *infinite-duration two-person games over graphs*, in particular *reachability* and *safety games*.

The following chapter cannot cover every aspect in complete detail. If necessary, we refer the reader to standard textbooks on automata theory and formal languages [HU79, HMU01], on first-order logics [KS08, BHvMW09], and on game theory [GTW02].

We begin with basic definitions and notations.

Basic Definitions and Notations Let $\mathbb{N} = \{0, 1, \ldots\}$ denote the set of natural numbers, $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ the set of natural numbers without 0, $\mathbb{Z}$ the set of integers, and $\mathbb{R}$ the set of real numbers. Moreover, for $k \in \mathbb{N}$ let $[k] = \{0, \ldots, k-1\}$ denote the set consisting of the first k natural numbers; in particular, $[0] = \emptyset$ holds.

Let X, Y, and Z be arbitrary sets. The set $2^X = \{Y \mid Y \subseteq X\}$ denotes the *power set of* X, and $Z = X \mathbin{\dot\cup} Y$ denotes the *disjoint union* of X and Y (i.e., $Z = X \cup Y$ and $X \cap Y = \emptyset$). For $n \in \mathbb{N}_+$, we use X^n as a shorthand-notation for the *n-fold Cartesian product of* X, defined by $X^n = \{(x_0, \ldots, x_{n-1}) \mid x_i \in X, i \in [n]\}$. A *binary relation* is a subset $R \subseteq X \times Y$. Given a binary relation $R \subseteq X \times Y$, the relation $R^{-1} = \{(v, u) \mid (u, v) \in R\}$ is the *inverse of* R. For a set $U \subseteq X$, the set $R(U) = \{y \in Y \mid \exists x \in U \colon (x, y) \in R\}$ is called the *image* of U under R. Similarly, for a set $V \subseteq Y$, the set $R^{-1}(V) = \{x \in X \mid \exists y \in V \colon (x, y) \in R\}$ is called the *preimage* of U under R. The *composition* of two binary relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ is the binary relation $R \circ S = \{(x, z) \in X \times Z \mid \exists y \in Y \colon (x, y) \in R \land (y, z) \in S\}$.

The *transitive closure* of a relation $R \subseteq X \times X$ is the relation $R^+ = \bigcup_{i \in \mathbb{N}_+} R^i$ where $R^1 = R$ and $R^{j+1} = R \circ R^j$ for $j \in \mathbb{N}_+$. The *reflexive and transitive closure* of R is the relation $R^* = R^+ \cup \{(x,x) \mid x \in X\}$.

An *equivalence relation* over a set X is a binary relation $\sim \subseteq X \times X$ that satisfies the following conditions for all $x, y, z \in X$:

- $(x,x) \in \sim$ for all $x \in X$ (*reflexivity*).
- If $(x,y) \in \sim$, then $(y,x) \in \sim$ (*symmetry*).
- If $(x,y) \in \sim$ and $(y,z) \in \sim$, then $(x,z) \in \sim$ (*transitivity*).

To ease notation, we use the infix notation $x \sim y$ rather than to write $(x,y) \in \sim$. The *equivalence class* of x is the set $[\![x]\!]_\sim = \{y \in X \mid x \sim y\}$. The *index* of an equivalence relation $\sim$ is defined by $\text{index}(\sim) = |\{[\![x]\!]_\sim \mid x \in X\}|$ (i.e., the number of its equivalence classes).

2.1 Finite Automata, Transducers, and Rational Graphs

We begin this section by fixing the notation of basics of formal language theory such as words, languages, and finite automata. Then, we extend the notion of finite automata to transducers, which do not process words but pairs of words. This allows us to define automatic graphs, which we use in Chapter 4 and Chapter 6 as finite representations of infinite graphs.

Words and Languages

An *alphabet* Σ is a nonempty, finite set of *symbols*. A *word* $u = a_1 \dots a_n$ is a finite sequence of symbols $a_i \in \Sigma$ for $i \in \{1, \dots, n\}$. The empty sequence is called the *empty word* and is denoted by ε. The *length* of a word u, denoted by $|u|$, is the number of symbols in u. Moreover, $|u|_a$ denotes the number of occurrences of the symbol a in u. For two words $u = a_1 \dots a_m$ and $v = b_1 \dots b_n$, the *concatenation of u and v* is the word $u \cdot v = uv = a_1 \dots a_m b_1 \dots b_n$. Furthermore, if $u = vw$, then v is a *prefix* of u and w is a *postfix* of u.

The set of all (finite) words over an alphabet Σ (i.e., the free monoid on Σ) is denoted by Σ^*. A subset $L \subseteq \Sigma^*$ is called a *language*. We lift the notion of concatenation from words to languages by defining the *concatenation of two languages L, L'* as $L \cdot L' = LL' = \{uv \mid u \in L, v \in L'\}$. Moreover, L^* is the Kleene iteration of L defined by $L^* = \bigcup_{i \in \mathbb{N}} L^i$ where $L^0 = \{\varepsilon\}$ and $L^{i+1} = L \cdot L^i$. The set of all prefixes of words in a language L is the set $\text{Pref}(L) = \{u \mid \exists v \in \Sigma^* \colon uv \in L\}$, and L is termed *prefix-closed* if $\text{Pref}(L) \subseteq L$.

Given a total order $\leq_\Sigma \subset \Sigma \times \Sigma$ over an alphabet Σ, the induced *canonical order on words* is the total order $\leq \subset \Sigma^* \times \Sigma^*$ that satisfies

$$a_1 a_2 \dots a_m \leq b_1 b_2 \dots b_n$$

if and only if

- $m = n$ and $a_1 a_2 \dots a_m = b_1 b_2 \dots b_n$;
- $m < n$; or
- $m = n$ and there exists an $i \in \{1, \dots, m\}$ such that $a_j = b_j$ for all $j \in \{1, \dots, i-1\}$, $a_i \neq b_i$, and $a_i \leq_\Sigma b_i$.

We denote the associated strict total order on words by $<$.

An *infinite word* over an alphabet Σ is an infinite sequence $\alpha = a_1 a_2 \dots$ of symbols $a_i \in \Sigma$ for $i \in \mathbb{N}_+$. The set of all infinite words over the alphabet Σ is denoted by Σ^ω. If $\alpha = u\beta$ for a $u \in \Sigma^*$ and $\beta \in \Sigma^\omega$, then we call u a *prefix* of α. For a (finite) word $u \in \Sigma^*$, the infinite word $u^\omega = uu \dots \in \Sigma^\omega$ is the infinite repetition of u. We refer the reader to Perrin and Pin [PP04] for an introduction to infinite words and automata operating on them.

Finite Automata

A *(nondeterministic) finite automaton (NFA)*[1] is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ where Q is a finite, nonempty set of *states*, Σ is the input alphabet, $q_0 \in Q$ is the distinguished *initial state*, $\Delta \subseteq Q \times \Sigma \times Q$ is the set of *transitions* (the *transition relation*), and $F \subseteq Q$ is the set of *final* states. A *run* of an NFA $\mathcal{A}$ from a state $q_1 \in Q$ on some word $u = a_1 \dots a_n \in \Sigma^*$ is a sequence $q_1, \dots, q_{n+1}$ of states $q_i \in Q$ such that $(q_i, a_i, q_{i+1}) \in \Delta$ for $i \in \{1, \dots, n\}$; we also write $\mathcal{A}\colon q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_{n+1}$ or $\mathcal{A}\colon q_1 \xrightarrow{u} q_{n+1}$ for short. A word u is *accepted* by $\mathcal{A}$ if $\mathcal{A}\colon q_0 \xrightarrow{u} q$ with $q \in F$. The language *accepted*—or *recognized*—by $\mathcal{A}$ is the set $L(\mathcal{A}) = \{u \in \Sigma^* \mid \mathcal{A}\colon q_0 \xrightarrow{u} q, q \in F\}$. A language $L \subseteq \Sigma^*$ is called *regular* if an NFA $\mathcal{A}$ with $L = L(\mathcal{A})$ exists. To measure the "complexity" of NFAs, we define the *size* of an NFA $\mathcal{A}$, denoted by $|\mathcal{A}|$, to be the number of its states (i.e., $|\mathcal{A}| = |Q|$).

A *deterministic finite automaton (DFA)* is an NFA where for every pair of state and input-symbol a unique destination-state exists. More formally, a DFA is an NFA where for every $p \in Q$ and $a \in \Sigma$ a transition $(p, a, q) \in \Delta$ exists and where $(p, a, q) \in \Delta$ and $(p, a, q') \in \Delta$ implies $q = q'$. In the case of DFAs, we substitute the transition relation Δ with a *transition function* $\delta\colon Q \times \Sigma \to Q$, which assigns to every pair of state and input-symbol a unique destination-state.

[1] We use the terms *finite automaton* and *nondeterministic finite automaton* synonymously in this thesis.

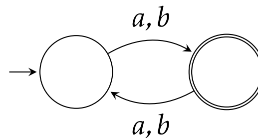


Figure 2.1: A DFA accepting the language $L = \{u \in \{a,b\}^* \mid |u| \text{ is odd}\}$.

Example 2.1. This simple example is meant to introduce our graphical notation of finite automata. Consider the DFA depicted in Figure 2.1, which accepts the language of all words over $\{a,b\}$ that have odd length. The state →◯ represents the initial state, and double-circled states ◎ represent final states. Transitions are depicted as arrows of the form $\bigcirc \xrightarrow{a} \bigcirc$. ◁

In a later chapter, we use *Moore machines*, which are DFAs that have no final states but are equipped with an output at every state. Formally, a Moore machine is a six tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta, \lambda)$ where Q, Σ, q_0, and δ are as in DFAs, Γ is the *output alphabet*, and $\lambda \colon Q \to \Gamma$ is the output function that maps each state to an output-symbol. A Moore machine does not accept a language but computes a mapping $f_\mathcal{M} \colon \Sigma^* \to \Gamma$ that maps an input-word $u \in \Sigma^*$ to the output of the unique state reached after reading u (i.e., $f_\mathcal{M}(u) = \lambda(q)$ if and only if $\mathcal{M} \colon q_0 \xrightarrow{u} q$). We call a function $f \colon \Sigma^* \to \Gamma$ *Moore machine-computable* if a Moore machine $\mathcal{M}$ with $f = f_\mathcal{M}$ exists. Note that we deliberately view a Moore machine as a device that maps an input-word to an output-symbol. A similar model in which Moore machines map an input-word to the sequence of outputs that occur during a run is also common but less suitable for algorithmic learning.

Myhill-Nerode Congruence

The *Myhill-Nerode congruence*—Nerode congruence for short—of a language $L \subseteq \Sigma^*$ is an equivalence relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ defined on words. Formally, two words $u, v \in \Sigma^*$ are *L-equivalent*, denoted by $u \sim_L v$, if $uw \in L \Leftrightarrow vw \in L$ is satisfied for all $w \in \Sigma^*$. For brevity, we denote the equivalence class of a word $u \in \Sigma^*$ by $[\![u]\!]_L = \{v \in \Sigma^* \mid u \sim_L v\}$ (rather than by $[\![u]\!]_{\sim_L}$). Recall that index of $\sim_L$ (i.e., the number of equivalence classes) is given by $\text{index}(\sim_L) = |\{[\![u]\!]_L \mid u \in \Sigma^*\}|$.

The Myhill-Nerode theorem (cf. Hopcroft and Ullman [HU79]) shows, among other things, that a language L is regular if and only if $\text{index}(\sim_L)$ is finite. Furthermore, it is not hard to verify that $\sim_L$ is a *congruence* with respect to concatenation; that is, if $u \sim_L v$, then $ua \sim_L va$ also holds for all $a \in \Sigma$.

The congruence property can be exploited to show that for every regular language L, there exists an up to isomorphism unique minimal DFA $\mathcal{A}_L$ that accepts L; this DFA is often called the *canonical minimal DFA for L*. The idea is to use the L-equivalence

classes as states of $\mathcal{A}_L$ and to exploit the congruence property of $\sim_L$ to define the transitions of $\mathcal{A}_L$. Formally, $\mathcal{A}_L = (Q_L, \Sigma, q_0^L, \delta_L, F_L)$ is defined by

- $Q_L = \{[\![u]\!]_L \mid u \in \Sigma^*\}$;
- $q_0^L = [\![\varepsilon]\!]_L$;
- $\delta_L([\![u]\!]_L, a) = [\![ua]\!]_L$; and
- $F_L = \{[\![u]\!]_L \mid u \in L\}$.

This definition is sound since $\sim_L$ is a congruence and $\mathrm{index}(\sim_L)$ is finite. Moreover, a standard induction over the length of input-words proves that $\mathcal{A}_L$ accepts L. Since $\mathrm{index}(\sim_L) \leq |\mathcal{A}|$ holds for every DFA $\mathcal{A}$ accepting L, the DFA $\mathcal{A}_L$ is indeed minimal among all DFAs accepting L. To prove that $\mathcal{A}_L$ is unique up to isomorphism, one defines an isomorphism that maps the states of some minimal DFA accepting L to the corresponding L-equivalence classes of $\mathcal{A}_L$.

Synchronous and Asynchronous Transducers

Transducer are automata that read pairs $(u,v) \in \Sigma^* \times \Gamma^*$ of words and accept binary relations $R \subseteq \Sigma^* \times \Gamma^*$ rather than languages. Another way of viewing transducers is that a transducer translates an input-word $u \in \Sigma^*$ into one or more output-words $v \in \Gamma^*$. Therefore, one often refers to Σ as the *input alphabet* and to Γ as the *output alphabet*.

Two different types of transducers are commonly used: synchronous transducers and asynchronous transducers.

Synchronous transducers A *synchronous transducer*, typically denoted by $\mathcal{T}$, is an NFA working over the alphabet $(\Sigma \cup \{\Box\}) \times (\Gamma \cup \{\Box\})$ where the symbol $\Box$ is neither contained in Σ nor in Γ. A synchronous transducer processes a pair $(u,v) \in \Sigma^* \times \Gamma^*$ by reading the so-called *convoluted word* $u \otimes v \in ((\Sigma \cup \{\Box\}) \times (\Gamma \cup \{\Box\}))^*$: formally, the *convolution* of two words $u = a_1 \ldots a_m$ and $v = b_1 \ldots b_n$ is the word $u \otimes v = (a'_1, b'_1) \ldots (a'_k, b'_k)$ of length $k = \max(m,n)$ where

$$a'_i = \begin{cases} a_i & \text{if } i \in \{1, \ldots, m\}; \\ \Box & \text{otherwise}; \end{cases} \quad \text{and } b'_i = \begin{cases} b_i & \text{if } i \in \{1, \ldots, n\}; \\ \Box & \text{otherwise}; \end{cases}$$

that is, $\Box$ is used as a padding symbol, which, if necessary, is appended to either u or v to balance the length of both words.

A synchronous transducer $\mathcal{T} = (Q, (\Sigma \cup \{\Box\}) \times (\Gamma \cup \{\Box\}), q_0, \Delta, F)$ *accepts* a pair of words $(u,v) \in \Sigma^* \times \Gamma^*$ if $\mathcal{T}: q_0 \xrightarrow{u \otimes v} q$ with $q \in F$ holds. Furthermore, $\mathcal{T}$ *defines*—or accepts—the binary relation $R(\mathcal{T}) = \{(u,v) \in \Sigma^* \times \Gamma^* \mid \mathcal{T}: q_0 \xrightarrow{u \otimes v} q, q \in F\}$. Finally, we

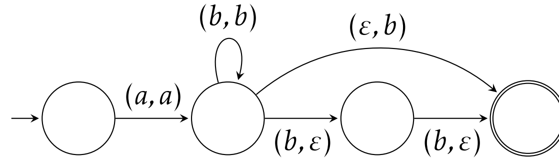


Figure 2.2: An asynchronous transducer $\mathcal{T}$ over the alphabets $\Sigma = \Gamma = \{a, b\}$ that defines the relation $R(\mathcal{T}) = \{(ab^n, ab^{n+1}) \mid n \geq 0\} \cup \{(ab^{n+2}, ab^n) \mid n \geq 0\}$.

call a binary relation $R \subseteq \Sigma^* \times \Gamma^*$ *automatic* if a synchronous transducer $\mathcal{T}$ with $R = R(\mathcal{T})$ exists.

Asynchronous transducers Asynchronous transducers—or simply *transducers* for short—are a generalization of synchronous transducers. Formally, an *asynchronous transducer* is a six-tuple $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ where Q is a finite, nonempty set of states, Σ is the input alphabet, Γ is the output alphabet, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$ is the transition relation, and $F \subseteq Q$ is the set of final states. We assume without loss of generality that an asynchronous transducer does not contain transitions of the form $(p, \varepsilon, \varepsilon, q)$; if necessary, such transitions can easily be eliminated (e.g., see Hopcroft and Ullman [HU79]).

A *run* of an asynchronous transducer $\mathcal{T}$ on a pair $(u, v) \in \Sigma^* \times \Gamma^*$ of words from a state $q_1 \in Q$ is a sequence of states $q_1, \ldots, q_{n+1}$ with $q_i \in Q$, $i \in \{1, \ldots, n\}$, such that there exist transitions $(q_i, u_i, v_i, q_{i+1}) \in \Delta$ with $u_i \in \{\varepsilon\} \cup \Sigma$, $v_i \in \{\varepsilon\} \cup \Gamma$, as well as $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_n$. In this case, we also write $\mathcal{T} : q_1 \xrightarrow{(u_1, v_1)} q_2 \xrightarrow{(u_2, v_2)} \cdots \xrightarrow{(u_n, v_n)} q_{n+1}$ or $\mathcal{T} : q_1 \xrightarrow{(u, v)} q_{n+1}$ for short. An asynchronous transducer *accepts* the pair (u, v) if $\mathcal{T} : q_0 \xrightarrow{(u, v)} q$ with $q \in F$. Intuitively, the difference to synchronous transducers is that asynchronous transducers can decide at any time to asynchronously process only one component of the input pair.

An asynchronous transducer $\mathcal{T}$ *defines*—or *accepts*—the binary relation $R(\mathcal{T}) = \{(u, v) \in \Sigma^* \times \Gamma^* \mid \mathcal{T} : q_0 \xrightarrow{(u, v)} q, q \in F\}$. Moreover, a binary relation $R \subseteq \Sigma^* \times \Gamma^*$ is called *rational* if an asynchronous transducer $\mathcal{T}$ with $R = R(\mathcal{T})$ exists. It is not hard to verify that each automatic relation is also rational because every synchronous transducer can be converted into an asynchronous one accepting the same relation. However, the converse is not true in general: a simple example for this fact is the infix relation $\{(uvw, v) \mid u, v, w \in \Sigma^*\} \subseteq \Sigma^* \times \Sigma^*$ for an alphabet Σ with $|\Sigma| \geq 2$.

Before we continue, let us illustrate these concepts with an example.

Example 2.2. The transducer $\mathcal{T}$ over the alphabets $\Sigma = \Gamma = \{a, b\}$ depicted in Figure 2.2 defines the relation $R(\mathcal{T}) = \{(ab^n, ab^{n+1}) \mid n \geq 0\} \cup \{(ab^{n+2}, ab^n) \mid n \geq 0\}$. ◁

Operations on Transducers Transducers allow performing many operations on the relation they represent symbolically. For instance, an operation often used in the context of Regular Model Checking is computing the image of a regular language under a rational (or automatic) relation. As the following lemma states, the result of this operation is again a regular language, and one can effectively construct an NFA accepting this language.

Lemma 2.1. *Let $\mathcal{A}$ be an NFA over the alphabet Σ and $\mathcal{T}$ an asynchronous transducer over the input alphabet Σ and the output alphabet Γ. Then, $R(\mathcal{T})(L(\mathcal{A}))$ is again a regular language, and one can effectively construct an NFA that accepts $R(\mathcal{T})(L(\mathcal{A}))$.*

This result is well-known, but since we use it in Chapter 6, the reader might find a brief proof sketch helpful.

Proof of Lemma 2.1. Let $\mathcal{T} = (Q_T, \Sigma, \Gamma, q_0^T, \Delta_T, F_T)$ be an asynchronous transducer and $\mathcal{A} = (Q_A, \Sigma, q_0^A, \Delta_A, F_A)$ an NFA. We prove Lemma 2.1 by constructing an NFA, which we call the *input-synchronous product* of $\mathcal{T}$ and $\mathcal{A}$, that accepts $R(\mathcal{T})(L(\mathcal{A}))$. To simplify matters, we construct this product in terms of an ε-NFA[2]. Then, one eliminates its ε-transitions (e.g., as described by Hopcroft and Ullman [HU79]) to obtain the desired NFA.

The underlying idea of this construction is to let the input-synchronous product guess the input $u \in L(\mathcal{A})$ of a pair (u, v) while reading the output v. Formally, the input-synchronous product of $\mathcal{T}$ and $\mathcal{A}$ is the ε-NFA $\mathcal{T} \cap \mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ defined by

- $Q = Q_T \times Q_A$;
- $q_0 = (q_0^T, q_0^A)$;
- $F = F_T \times F_A$; and
- $\Delta = \{((p,p'), b, (q,q')) \mid \exists a \in \Sigma \colon (p,(a,b),q) \in \Delta_T, (p',a,q') \in \Delta_A\} \cup$
  $\{((p,p'), \varepsilon, (q,q')) \mid \exists a \in \Sigma \colon (p,(a,\varepsilon),q) \in \Delta_T, (p',a,q') \in \Delta_A\} \cup$
  $\{((p,p'), b, (q,p')) \mid (p,(\varepsilon,b),q) \in \Delta_T\}$.

A standard induction over the number of transitions used in a run of $\mathcal{T} \cap \mathcal{A}$ proves that $L(\mathcal{T} \cap \mathcal{A}) = R(\mathcal{T})(\mathcal{A})$ indeed holds. □

A simple observation is that one obtains $R(\mathcal{T})^{-1}$ by swapping the input and output component of a transducer. Thus, one immediately obtains the following corollary by applying the construction from above to the transducer defining $R(\mathcal{T})^{-1}$.

[2] An ε-NFA is an NFA whose transition relation is a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

Corollary 2.1. Let $\mathcal{A}$ be an NFA over the alphabet Γ and $\mathcal{T}$ an asynchronous transducer over the input alphabet Σ and the output alphabet Γ. Then, $R(\mathcal{T})^{-1}(L(\mathcal{A}))$ is again a regular language, and one can effectively construct an NFA that accepts $R(\mathcal{T})^{-1}(L(\mathcal{A}))$.

Finally, let us mention that a similar construction shows that both Lemma 2.1 and Corollary 2.1 also hold for synchronous transducers.

Automatic and Rational Graphs

A *(directed) graph* is a tuple $G = (V, E)$ where V is a countable set of *vertices* and $E \subseteq V \times V$ is a set of directed *edges*. If V is a finite set, we call G a *finite graph*; if V is an infinite set, we also call G an *infinite graph* to emphasize this fact.

To work with infinite graphs algorithmically, one needs a finite representation. Since we are interested in applying techniques for learning regular languages, we follow an approach described, amongst others, by Blumensath and Grädel [BG04] and represent (infinite) graphs by means of finite automata and transducers. The idea behind this representation is to provide a surjective function $\nu\colon V \to \Sigma^*$, which associates each vertex $v \in V$ uniquely with a finite word $u \in \Sigma^*$ taken from some fixed alphabet Σ, such that

- $L_V = \{\nu(v) \in \Sigma^* \mid v \in V\}$ is a regular language representing the vertices; and
- $R_E = \{(\nu(v), \nu(v')) \in L_V \times L_V \mid (v, v') \in E\}$ is a rational relation representing the edges.

Formally, we call a graph $G = (V, E)$ *rational* if there exists a surjective function $\nu\colon V \to \Sigma^*$ for a suitable alphabet Σ such that the language L_V is regular and the relation R_E is rational. Each rational graph can be represented by a tuple $(\mathcal{A}, \mathcal{T})$ where $\mathcal{A}$ is an NFA (or DFA) that accepts L_V and $\mathcal{T}$ is an asynchronous transducer $\mathcal{T}$ that defines R_E; for brevity, we then also write $G = (\mathcal{A}, \mathcal{T})$. This representation is what we use later as inputs to our algorithms.

A simple case occurs if $V \subseteq \Sigma^*$ itself is a regular language and $E \subseteq V \times V$ is a rational relation. In this case, the graph $G = (V, E)$ is clearly rational because we can choose ν to be the identity, which entails $L_V = V$ and $L_E = E$. Let us illustrate this with an example.

Example 2.3. Reconsider the asynchronous transducer $\mathcal{T}$ of Example 2.2 depicted in Figure 2.2. The graph $G = (V, E)$ with $V = ab^*$ and $E = R(\mathcal{T})$ is rational. It is sketched in Figure 2.3. ◃

An important subclass of rational graphs forms the class of automatic graphs, which are defined in terms of synchronous rather than asynchronous transducers.

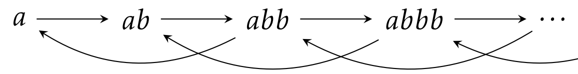


Figure 2.3: The rational graph $G = (V, E)$ with $V = ab^*$ and $E = R(\mathcal{T})$ where $\mathcal{T}$ is the asynchronous transducer of Figure 2.2.

Consequently, we call a graph $G = (V, E)$ *automatic* if a surjective function $\nu\colon V \to \Sigma^*$ exists such that L_V is regular and R_E is automatic.

Finally, let us summarize a few observations about automatic and rational graphs, to which we refer in later chapters.

Observation 2.1. Every finite graph $G = (V, E)$ is automatic and, hence, also rational because we can choose $\Sigma = V$ and ν to be the identity. Then, L_V is a finite language and, therefore, regular. Furthermore, R_E is a finite relation and, hence, automatic.

Observation 2.2. The *reachability problem* for automatic (respectively rational) graphs, which is the decision problem

> "Given an automatic (respectively rational) graph $G = (V, E)$ and two vertices $v, v' \in V$. Does $v' \in E^*(\{v\})$ hold?",

is undecidable. The intuitive reason is that automatic graphs are already expressive enough to encode runs of Turing machines. When viewed from this perspective, the reachability problem corresponds to the halting problem of Turing machines, which is known to be undecidable (e.g., see Papadimitriou [Pap94]). We refer the reader to Thomas [Tho01] for further details.

Observation 2.3. If $G = (\mathcal{A}, \mathcal{T})$ is an automatic (or rational) graph and $L \subseteq L(\mathcal{A})$ a regular language containing a subset of vertices, then both the set $R(\mathcal{T})$ of all successors of vertices in L and the set $R(\mathcal{T})^{-1}$ of all predecessors of vertices in L are regular. In addition, one can effectively construct NFAs that accept the respective sets. This observation is a simple corollary of Lemma 2.1 and Corollary 2.1.

2.2 First-order Logics

This section is a brief introduction to first-order logic (containing an excursus to a monadic second-order logic). We assume a basic understanding of this subject and introduce the following concepts with a reader in mind who is mainly interested in applying logics and decision procedures rather than in their internals (i.e., we omit details at some points in favor of a more accessible description). For a thorough investigation of this subject, we refer to textbooks on this topic [KS08, BHvMW09]. The following definitions loosely follow the description by de Moura and Bjørner [dMB09].

First-order logic is based on three elements:

- A set of variables
- A signature that contains function symbols and predicate symbols
- A syntax that defines how formulas have to be constructed from variables, symbols of the signature, and logical symbols such as $\neg$, $\wedge$, quantifiers, etc.

A formula is evaluated in a so-called interpretation (which we define shortly). Intuitively, an interpretation determines the domain of the variables, assigns values to the variables, and fixes the semantics of the symbols of the signature. By restricting parts of an interpretation (e.g., by restricting variables to the Boolean values *true* and *false*) or by disallowing certain logical symbols, we can "instantiate" different types of first-order logic.

At the end of this section, we introduce a number of restricted logics that we apply in later chapters of this thesis. This list comprises Propositional Boolean Logic, Equality Logic, Presburger arithmetic, a logic of uninterpreted functions over the natural numbers, and the Array Property Fragment. We also briefly discuss the logic Strand, which is not a first-order logic but a restricted type of monadic second order logic. All of these logics allow for effective decision procedures.

Let us first introduce the syntax and the semantics of first-order logic.

Syntax of First-order Logic

A signature defines the building blocks from which formulas are constructed. Formally, a *signature*[3] is a tuple $\Sigma = (S, F, P)$ consisting of a set S of *sorts*, a set F of *function symbols*, and a set P of *predicate symbols*; for technical reasons, we assume without loss of generality that $F \cap P = \emptyset$ is always satisfied. Intuitively, a sort is an abstract symbol that stands for the "type" of a variable, an argument of a function, and so on—we encourage the reader to have the Boolean values *true* and *false*, the natural number $\mathbb{N}$, etc. in mind. Each function symbol $f \in F$ is equipped with an *arity* of the form $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ where $n \in \mathbb{N}$, $\sigma \in S$, and $\sigma_i \in S$ for $i \in \{1, \ldots, n\}$; if $n = 0$, we call f a *constant*. Similarly, each predicate symbol $p \in P$ is equipped with an arity of the form $\sigma_1 \times \ldots \times \sigma_n$ where $n \in \mathbb{N}_+$; note that we require $n \geq 1$ in the case of predicate symbols.

Formulas in first-order logic are constructed from variables and terms:

- A *variable* x is taken from a countable set X and associated with a sort $\sigma \in S$.

[3] We follow the common notation in the literature and denote signatures by the symbol Σ. Since this section is self-contained, there is no danger of confusing signatures and alphabets, which we also denote by Σ.

- A *term* t with sort $\sigma \in S$ is either a variable $x \in X$ with sort σ or of the form $f(t_1,\dots,t_n)$ where $t_1,\dots,t_n$ are terms with sort $\sigma_i \in S$ for $i \in \{1,\dots,n\}$ and $f \in F$ is a function symbol with arity $\sigma_1 \times \dots \times \sigma_n \to \sigma$.

Given a signature Σ and a set X of variables, the syntax of *first-order formulas* over Σ and X—*Σ-formulas* for short—is defined inductively as follows:

- *true* and *false* are Σ-formulas.
- If $t_1,\dots,t_n$ are terms with sorts $\sigma_1,\dots,\sigma_n$ and $p \in P$ is a predicate symbol with arity $\sigma_1 \times \dots \times \sigma_n$, then $p(t_1,\dots,t_n)$ is a Σ-formula.
- If φ_1,φ_2 are two Σ-formulas, so are $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, and $(\varphi_1 \vee \varphi_2)$.
- If φ is a Σ-formula, so are $\exists x.\sigma\colon \varphi$ and $\forall x.\sigma\colon \varphi$ where $x \in X$ is a variable of sort $\sigma \in S$.

We add "syntactic sugar" and additionally allow formulas of the form $(\varphi_1 \to \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$. These formulas can be syntactically replaced by $(\varphi_1 \to \varphi_2) \coloneqq (\neg\varphi_1 \vee \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2) \coloneqq ((\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1))$.

For the sake of readability, we often omit parenthesis if they are clear from the context. Moreover, whenever common function symbols or predicate symbols, such as $+$ and $-$, occur, we use the natural infix notation; for instance, we write $x+y$ rather than $+(x,y)$.

A formula φ' is called a *subformula* of a formula φ if it occurs as an infix in φ (i.e., if it has been used in the inductive construction of φ). If φ does not contain a subformula, it is called an *atomic formula*. Moreover, a formula is called *quantifier free* if none of its subformulas is of the form $\exists x.\sigma\colon \varphi$ or $\forall x.\sigma\colon \varphi$.

A variable $x \in X$ can either occur *free* in a formula or *bound* to a quantifier (e.g., as $\exists x.\sigma\colon \varphi$ or $\forall x.\sigma\colon \varphi$). We denote the set of free variables of a formula φ by $\mathrm{free}(\varphi)$. If the free variables in φ are of special interest, we write $\varphi(x_1,\dots,x_n)$. We also use the more compact notation $\varphi(\overline{x})$ where $\overline{x} = (x_1,\dots,x_n)$ is a list of variables that is either defined explicitly or clear from the context. A formula without free variables is called a *sentence*.

Semantics of First-order Logic

A formula is evaluated in an interpretation. Given a signature $\Sigma = (S,F,P)$ and a set X of variables, an *interpretation* for Σ and X is a tuple $\mathcal{I} = ((D_\sigma)_{\sigma\in S},\beta)$ where D_σ is a nonempty set for every $\sigma \in S$ called the *domain* of the sort σ and β is the so-called *interpretation function*. For every sort $\sigma \in S$, the domain D_σ contains the values that variables of sort σ, functions of arity $\sigma_1 \times \dots \times \sigma_n \to \sigma$, and so on can assume. On the other hand, the interpretation function is a mapping that

- assigns to each variable $x \in X$ with sort σ an element $x^{\mathcal{I}} \in D_\sigma$;
- assigns to each constant $f \in F$ with sort σ an element $f^{\mathcal{I}} \in D_\sigma$;
- assigns to each function symbol $f \in F$ with arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ and $n > 0$ a total function $f^{\mathcal{I}} \colon D_{\sigma_1} \times \ldots \times D_{\sigma_n} \to D_\sigma$; and
- assigns to each predicate symbol $p \in P$ with arity $\sigma_1 \times \ldots \times \sigma_n$ and $n > 0$ a relation $p^{\mathcal{I}} \subseteq D_{\sigma_1} \times \ldots \times D_{\sigma_n}$.

We additionally lift interpretations to terms: the interpretation $t^{\mathcal{I}}$ of a term t is either $x^{\mathcal{I}}$ if $t = x$ or $f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$ if $t = f(t_1, \ldots, t_n)$. Finally, we define $\mathcal{I}\{x \leftarrow d\}$ to be the interpretation in which the variable x with sort σ is fixed to the value $d \in D_\sigma$.

The semantics of Σ-formulas is defined in terms of the satisfaction relation $\models$. Given a Σ-formula over the set X of variables and an interpretation $\mathcal{I}$ for Σ and X, *satisfiability*, denoted by $\mathcal{I} \models \varphi$, is inductively defined as follows:

$$\begin{aligned}
\mathcal{I} \models \mathit{true} &\text{ and } \mathcal{I} \not\models \mathit{false}. \\
\mathcal{I} \models p(t_1, \ldots, t_n) &\text{ if and only if } (t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in p^{\mathcal{I}}. \\
\mathcal{I} \models \neg\varphi &\text{ if and only if } \mathcal{I} \not\models \varphi. \\
\mathcal{I} \models (\varphi_1 \land \varphi_2) &\text{ if and only if } \mathcal{I} \models \varphi_1 \text{ and } \mathcal{I} \models \varphi_2. \\
\mathcal{I} \models (\varphi_1 \lor \varphi_2) &\text{ if and only if } \mathcal{I} \models \varphi_1 \text{ or } \mathcal{I} \models \varphi_2. \\
\mathcal{I} \models \exists x.\sigma \colon \varphi &\text{ if and only if } \mathcal{I}\{x \leftarrow d\} \models \varphi \text{ for some } d \in D_\sigma. \\
\mathcal{I} \models \forall x.\sigma \colon \varphi &\text{ if and only if } \mathcal{I}\{x \leftarrow d\} \models \varphi \text{ for all } d \in D_\sigma.
\end{aligned}$$

A Σ-formula φ over the set X of variables is called *satisfiable* if there exists an interpretation $\mathcal{I}$ for Σ and X such that $\mathcal{I} \models \varphi$; in this case, we call $\mathcal{I}$ a *model* of φ and use the special symbol $\mathfrak{M}$ to emphasize that the interpretation is a model. We call two Σ-formulas φ_1 and φ_2 over the same set X of variables *equivalent* if every interpretation $\mathcal{I}$ over Σ and X satisfies the equivalence

$$\mathcal{I} \models \varphi_1 \text{ if and only if } \mathcal{I} \models \varphi_2.$$

Let us illustrate these definitions with an example.

Example 2.4. Consider the signature $\Sigma = (\{\sigma\}, \{+\}, \emptyset)$ where $+$ has the arity $\sigma \times \sigma \to \sigma$. Moreover, let

$$\varphi(y) \coloneqq \exists x.\sigma \colon x + y = x$$

be a Σ-formula ranging over variables from $X = \{x, y\}$; since σ is the only sort, both x and y necessarily have the sort σ. The variable y is a free variable whereas x is not.

The interpretation $\mathcal{I}_1 = (D_\sigma, \beta)$ where $D_\sigma = \mathbb{N}$ and β maps x to the value 1, y to the value 0, and + to the usual addition of natural numbers is a model of φ. On the other hand, the interpretation $\mathcal{I}_1 = (D_\sigma, \beta)$ where $D_\sigma = \mathbb{N} \setminus \{0\}$ and β maps x and y to arbitrary values is not a model of φ. ◁

From an application point of view, we want to translate a task of an application domain into a first-order formula φ such that a model of φ (i.e., an interpretation of variables as well as function and predicate symbols) can be retranslated into a solution of the original problem. Thus, we are interested in solving the decision problem

"Given a Σ-formula φ over a set X of variables. Is φ satisfiable?",

which is known as the *satisfiability problem of first-order logic*—*satisfiability problem* for short. We require a decision procedure for the satisfiability problem to provide a model for φ provided the formula is satisfiable.[4] The process of finding a satisfying solution is commonly referred to as *solving* a formula and, hence, decision procedures are often termed *solvers*.

Restrictions of First-order Logic

First-order logic is expressive enough to encode runs of Turing machines, and the satisfiability problem is undecidable in general [BGG01].[5] Thus, first-order logic needs to be restricted in order to permit the construction of decision procedures. In slight abuse of terminology, these restrictions are often also called logics.

To make the following presentation of restricted logics more accessible and more self-contained, we introduce these restrictions whilst taking the following conventions into account:

- We provide a separate syntax (if necessary), which then implicitly specifies the signature. We mention explicitly whenever additional function or predicate symbols are allowed.
- We explicitly fix the domains and the interpretation of some or all function and predicate symbols.
- We often omit the sort of variables and arities if these are clear from the context. In particular, we always omit arities if a logic is constructed over a signature that contains a single sort.

[4] This slightly different problem, which entails the satisfiability problem, is called *constraint satisfaction problem*.
[5] This result is known as *Trakhtenbrot's theorem*.

One can easily lift all notions defined for first-order logic, such as subformulas and free variables, to restricted logics. In particular, a Σ-formulas φ of a restricted logic is *satisfiable* if there exists a restricted interpretation $\mathcal{I}$ for Σ and X satisfying $\mathcal{I} \models \varphi$. Moreover, two restricted Σ-formulas φ_1 and φ_2 over the same set of variables are *equivalent* if all interpretations $\mathcal{I}$ taken from the restricted class of interpretations satisfy $\mathcal{I} \models \varphi_1$ if and only if $\mathcal{I} \models \varphi_2$.

Let us now introduce the logics that we use later in this thesis.

Propositional Boolean Logic

Propositional Boolean Logic is the single-sorted quantifier-free first-order logic over the signature

$$\Sigma_{\mathrm{PL}} = (\{\sigma_{\mathbb{B}}\}, \emptyset, \emptyset),$$

where the domain of the sort $\sigma_{\mathbb{B}}$ is fixed to the set $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ of Boolean values.

The main building blocks of formulas in Propositional Boolean Logic are *literals*: a literal is either a variable $x \in X$ or its negation $\neg x$. The syntax of formulas in Propositional Boolean Logic is given as follows:

- Each literal is a formula.
- If φ_1 and φ_2 are two formulas, so are $\neg\varphi_1$, $(\varphi_1 \land \varphi_2)$, and $(\varphi_1 \lor \varphi_2)$.

Additionally, we add syntactic sugar and allow formulas of the form $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$; the formula $(\varphi_1 \rightarrow \varphi_2)$ is the shorthand-notation for $(\neg\varphi_1 \lor \varphi_2)$ whereas $(\varphi_1 \leftrightarrow \varphi_2)$ is the shorthand-notation for $(\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1)$. Moreover, we allow the formulas *true* and *false*, which can be expressed by $(x \lor \neg x)$ and $(x \land \neg x)$.

Note that this syntax does not completely comply with the syntax of first-order logic because variables need to occur as arguments of predicate symbols. However, we follow the usual convention in the literature and extend the satisfaction relation by defining $\mathcal{I} \models x$ if and only if $x^{\mathcal{I}} = \mathit{true}$ and $\mathcal{I} \models \neg x$ if and only if $x^{\mathcal{I}} = \mathit{false}$ (rather than introducing a special predicate symbol "true(x)").

The satisfiability problem for Proposition Boolean Logic is an important problem in theory and practice because many other problems can be reduced to it. Due to its frequent use, the acronym *SAT* has been established in the literature, and decision procedures for SAT are typically referred to as *SAT solvers*. For the rest of this thesis, we also follow this naming convention.

Although it is well known that the satisfiability problem for formulas in Proposition Boolean Logic is NP-complete [Coo71], the theoretical and practical importance of this logic has led to powerful SAT solvers. In fact, these solvers are able to solve

many real-world problems with hundreds of thousands or even millions of variables reasonably efficient. Prominent examples are MiniSat [ES03], Glucoser (a solver that extends the SAT solver Glucose [AS09] with extended resolution), and Sat4J [BP10]. These solvers were developed in academia and are freely available as open source.

Virtually all SAT solvers require their input to be in conjunctive normal form. A formula φ is in *conjunctive normal form (CNF)* if is of the form

$$\varphi := \bigwedge_{i=1}^{n} c_i \text{ with } c_i := \bigvee_{j=1}^{m_i} l_{ij},$$

where $n, m_1, \ldots, m_n \in \mathbb{N}_+$ are natural numbers and l_{ij} is a literal. The formulas c_i are called *clauses*.

Note that CNF as input format does not pose a restriction since one can efficiently convert every formula in Propositional Boolean Logic into one in CNF that preserves satisfiability (i.e., that is satisfiable if and only if the original formula is). A popular conversion is *Tseitin's encoding* (see [KS08] for a brief discussion), which introduces new variables and new clauses whose numbers are both linear in the number of subformulas occurring in the original formula.

Equality Logic

Equality Logic can be thought of as propositional logic in which the atoms are equalities between variables. More formally, *Equality Logic* is the single-sorted quantifier-free first-order logic over the signature

$$\Sigma_{\mathrm{EQ}} = (\{\sigma\}, \emptyset, \{=\}).$$

The domain D_σ might be any set, and the predicate symbol $=$ is always interpreted as the identity relation over D_σ.

The syntax of formulas in Equality Logic is as follows:

- If $x, y \in X$ are variables, then $x = y$ is a formula.
- If φ_1 and φ_2 are two formulas, so are $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, and $(\varphi_1 \vee \varphi_2)$.

Additionally, we allow formulas of the form $x \neq y$, which is a shorthand-notation for $\neg(x = y)$.

It is not hard to verify that a formula ranging over n variables can be satisfied by assigning at most n distinct values to the variables provided that it is satisfiable at all. Thus, if a formula in Equality Logic is satisfiable, it can be satisfied by a model whose domain is $[n]$ for a suitable $n \in \mathbb{N}$. We exploit this fact later in Section 3.1.1.

The satisfiability problem of formulas in Equality Logic is known to be NP-complete and can be reduced to the satisfiability problem of formulas in Propositional Boolean Logic (e.g., see Kroening and Strichman [KS08]). Nonetheless, *satisfiability modulo theory solvers (SMT solvers)*, such as Microsoft's Z3 [dMB08] and CVC [BCD+11], implement powerful decision procedures, which can solve even real-world instances in a reasonable amount of time (see the SMT competition [BDdM+13] for an overview).

Finally, let us mention that Equality Logic can also be defined to contain formulas of the form $x = c$ for a variable $x \in X$ and an arbitrary constant $c \in D_\sigma$. This, of course, means to abandon the property that we can choose values from the domain freely. For technical reasons, we do not consider this extension here.

Presburger Arithmetic

Presburger arithmetic is the first-order logic over the signature

$$\Sigma_{\mathrm{PA}} = (\{\sigma_\mathbb{Z}\}, \{+, -, 0, 1\}, \{<, =\})$$

in which

- the domain $D_{\sigma_\mathbb{Z}}$ is fixed to be the set of integers (i.e., $D_{\sigma_\mathbb{Z}} = \mathbb{Z}$);
- $+$ and $-$ are binary function symbols interpreted as the usual addition, respectively subtraction;
- 0 and 1 are constant symbols interpreted as $0 \in \mathbb{Z}$ and $1 \in \mathbb{Z}$;
- $<$ is a binary predicate symbols interpreted as the less-than relation, and $=$ is a binary predicate symbols interpreted as the identity relation.

Besides the constants 0 and 1, we additionally allow formulas to contain arbitrary constants $c \in \mathbb{Z}$ (which can syntactically be replaced by terms of the form $1 + 1 + \cdots + 1$ or $0 - 1 - \cdots - 1$). Moreover, we add "syntactic sugar" and allow the binary predicates $\neq$, $\leq, >$, and $\geq$ with their usual meaning. However, note that Presburger arithmetic does neither allow uninterpreted functions nor uninterpreted predicates.

Example 2.5. The formula

$$\forall x \colon \big(\exists y \colon (x \leq y - 1) \land \exists y \colon (x \geq y + 1)\big)$$

is a formula in Presburger arithmetic and states that $\mathbb{Z}$ contains infinitely many positive and negative integers. ◁

There exists a tight connection between formulas in Presburger arithmetic and regular languages. In fact, every formula $\varphi(x_1, \ldots, x_n)$ in Presburger arithmetic can

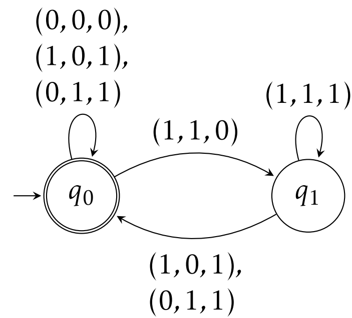


Figure 2.4: The DFA $\mathcal{A}^\varphi$ accepting the set of words that encode integers satisfying the formula $\varphi(x,y,z) := x+y=z$. Missing transitions lead to a nonaccepting sink-state, which is not shown for the sake of readability.

be associated with a DFA $\mathcal{A}^\varphi$ working over the alphabet $\{0,1\}^n$. The key idea is to interpret the sequence of bits obtained from concatenating the i-th component of an input-word as the value of the variable x_i (e.g., in a least-bit-first encoding). The DFA $\mathcal{A}^\varphi$ is constructed such that it accepts exactly those words that encode integers satisfying φ. We skip a formal construction here and refer the reader to Leroux [Ler05] for further details. Nonetheless, let us illustrate the main ideas of this construction through a short example.

Example 2.6. Consider the formula

$$\varphi(x,y,z) := x+y=z$$

over the variables x,y,z.

The input alphabet of the corresponding DFA $\mathcal{A}^\varphi$ contains symbols of the form (i,j,k) with $i,j,k \in \{0,1\}$. The first component of an input-word represents the bits encoding the variable x, the second component the bits encoding the variable y, and the third component those encoding the variable z; for instance, the input-word $(0,1,0)(0,1,0)(0,0,1)$ encodes the values $x=0$, $y=3$, and $z=4$.

The DFA $\mathcal{A}^\varphi$ is depicted in Figure 2.4. It performs an addition of x and y in binary and verifies that z corresponds to the result of this addition. To this end, it uses the states q_0 and q_1 to remember a carry (q_0 indicates no carry, whereas q_1 indicates a carry). If $\mathcal{A}^\varphi$ detects that the third component of the input does not equal the sum of the first two, it switches to a nonaccepting sink-state, which is omitted in the figure for the sake of readability. ◁

Note that not every regular language corresponds to a formula in Presburger arithmetic. However, Leroux [Ler05] showed that one can decide whether the language of a DFA is expressible in Presburger arithmetic in time polynomial in the size of the input

DFA. Moreover, if the language accepted by the given DFA is expressible by a formula in Presburger arithmetic, then such a formula can be computed in time polynomial in the size of the DFA as well.

The translation of Presburger formulas into DFAs can be used to decide the satisfiability problem: once a formula φ has been translated into the automata $\mathcal{A}^\varphi$, it is indeed enough to check whether the language $L(\mathcal{A}^\varphi)$ is empty. The Mona tool [HJJ+95] follows this approach.[6]

However, a naïve implementation of the automata-based approach results in a nonelementary time complexity. A more efficient approach has been proposed by Oppen [Opp78], which is based on quantifier elimination and decides the satisfiability of a sentence of length n in time $2^{2^{2^{cn}}}$ for a suitable constant $c > 1$. As lower bound, Fisher and Rabin [FR74] showed that any deterministic decision procedure has at least a doubly exponential worst-case runtime.

Logic of Uninterpreted Functions over the Natural Numbers

The *logic of uninterpreted functions over the natural numbers* is the quantifier-free fragment of first-order logic over the signature

$$\Sigma_{\mathrm{UF}} = (\{\sigma_{\mathbb{N}}\}, F, P \cup \{<, =\}),$$

where the domain $D_{\sigma_{\mathbb{N}}}$ is fixed to be the set of natural numbers, $<$ is a binary relation symbol interpreted as the less-than relation over the natural numbers, and $=$ is a binary predicate symbol interpreted as the identity relation over the natural numbers. The sets F and P may contain any number of function and predicate symbols (but P must not contain $<, =$) and are used to parameterize the logic. We refer to these symbols as *uninterpreted functions* and *uninterpreted predicates*, respectively. For ease of notation, we denote a function symbol f with arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ by $f : \mathbb{N}^n \to \mathbb{N}$ (note that $\sigma_1, \ldots, \sigma_n, \sigma$ are the same sort $\sigma_{\mathbb{N}}$).

Formulas in this logic are constructed as follows:

- A term is either a variable $x \in X$ or of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $f \in F$ is function symbol with $f : \mathbb{N}^n \to \mathbb{N}$.
- If $p \in P$ is a predicate symbol with arity $\sigma_1 \times \ldots \times \sigma_n$ and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula. In particular, if t_1, t_2 are terms, then $t_1 = t_2$ and $t_1 < t_2$ are formulas.
- If φ_1 and φ_2 are two formulas, so are $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, and $(\varphi_1 \vee \varphi_2)$.

[6] Originally, Mona has been designed for weak monadic second order logic over the natural numbers and the infinite binary tree, but also features native support for Presburger arithmetic.

We additionally allow arbitrary constants $c \in \mathbb{N}$ as well as the binary predicates $\neq$, $\leq, >$, and $\geq$ with their usual meaning.

As notational convention, we denote a formula by $\varphi(x_1, \dots, x_\ell, f_1, \dots, f_m, p_1, \dots, p_n)$ if $\text{free}(\varphi) = \{x_1, \dots, x_\ell\}$, $F = \{f_1, \dots, f_m\}$, and $P = \{p_1, \dots, p_n\}$. Moreover, we often specify the sets F and P only implicitly by the function and predicate symbols that occur in the formula.

Example 2.7. A simple example of a formula in the logic of uninterpreted functions over the natural numbers is

$$\varphi(x, f) := f(x) > x.$$

This formula ranges over a single free variable x and the uninterpreted function $f : \mathbb{N} \to \mathbb{N}$. The formula φ is satisfiable, and a model is, for instance, an interpretation $\mathfrak{M}$ with $x^{\mathfrak{M}} = 0$ and $f^{\mathfrak{M}}(x) = x + 1$. ◁

The logic of uninterpreted functions over the naturals is a fragment of the combination of two popular first-order logics, namely *linear arithmetic over the naturals* and *Equality Logic with uninterpreted functions*. The satisfiability problem for both logics is decidable (e.g., using the simplex algorithm for the former and a reduction to Propositional Boolean Logic for the latter), and a combination technique such as the *Nelson-Oppen procedure* provides a decision procedure for the satisfiability problem of formulas in the combined logic.[7] In fact, modern SMT solvers, such as Z3 and CVC, are capable of solving formulas in the logic of uninterpreted functions over the naturals and provide a model if the given formula is satisfiable.

Array Property Fragment

We use the *Array Property Fragment (APF)*, introduced by Bradley, Manna, and Sipma [BMS06], to reason about arrays in this thesis. The array property fragment is a decidable syntactic fragment of a more general (and undecidable) logic called *array logic*. We here introduce the Array Property Fragment in a way that fits our definition of first-order logic and loosely follows the description by Kroening and Strichman [KS08]. We refer the reader to Bradley, Manna, and Sipma [BMS06] as well as Kroening and Strichman [KS08] for further details about the array logic and the original definition of the Array Property Fragment.

The Array Property Fragment combines an *index logic*, which is used to express relations between array indices, and an *element logic*, which is used to express prop-

[7] We refer the reader to Kroening and Strichman [KS08] for a detailed description of linear arithmetic, quality logic and uninterpreted functions, as well as the Nelson-Oppen combination procedure.

erties of the elements in an array.[8] The index logic is fixed to be a slightly restricted version of Presburger arithmetic. More precisely, the index logic has the signature $\Sigma_I = (\{\sigma_\mathbb{Z}\}, \{0, 1, +, -\}, \{\leq, =\})$ where the function and predicate symbols are interpreted as in Presburger arithmetic (the Array Property Fragment uses $\leq$ with the usual meaning rather than $<$) and the domain of the sort $\sigma_\mathbb{Z}$ is fixed to be the set $\mathbb{Z}$. To reason about array entries, we assume that a decidable first-order logic over some signature $\Sigma_E = (\{\sigma_E\}, F_E, P_E)$ with $\{0, 1, +, -\} \not\in F_E$ and $\{\leq, =\} \not\in P_E$ is provided.

The Array Property Fragment is a multi-sorted first-order logic that uses the sort $\sigma_\mathbb{Z}$ to represent array indices and the sort σ_E to represent array elements. In addition, a third sort σ_A is used to represent arrays. The reader should interpret an array as a function $a \colon \mathbb{Z} \to D_{\sigma_E}$ that maps index positions to array entries of the domain D_{σ_E}. Accordingly, we fix the domain of the sort σ_A to be the set $D_{\sigma_A} = \{a \mid a \colon \mathbb{Z} \to D_{\sigma_E}\}$ (i.e., the set of all functions that map natural numbers to elements of D_{σ_E}).

The signature of the Array Property Fragment is

$$\Sigma_{\text{APF}} = (\{\sigma_A, \sigma_E, \sigma_\mathbb{Z}\}, F_E \cup \{[\,], \leftarrow, 0, 1, +, -\}, P_E \cup \{\leq, =\}),$$

where we additionally assume that $\{[\,], \leftarrow\} \not\in F_E$. The symbols $[\,]$ and $\leftarrow$ are two new function symbols that correspond to a read and write operation on the array, respectively. Formally, the function symbol $[\,]$ has arity $\sigma_A \times \sigma_\mathbb{Z} \to \sigma_E$ and corresponds to a read operation that returns the element of an array a at some index i; for ease of notation, we then write $a[i]$. The function symbol $\leftarrow$ has arity $\sigma_A \times \sigma_\mathbb{Z} \times \sigma_E \to \sigma_A$ and corresponds to a write operation that stores the value e at some index i in the array a; we denote this by $a\{i \leftarrow e\}$.

One formalizes the meaning of read and write operations by restricting the allowed interpretations; that is, one exclusively allows interpretations that satisfy the so-called *read-over-write axiom*:

$$\forall a \in D_{\sigma_A} \colon \forall e \in D_{\sigma_E} \colon \forall i, j \in \mathbb{Z} \colon a\{i \leftarrow e\}[j] = \begin{cases} e & i = j, \\ a[j] & \text{otherwise;} \end{cases}$$

that is, after the value e has been written into the array a at index i, the array entry at position i is the element e and all other array entries remain unchanged.

An *array property* is a formula of the form

$$\forall i_1.\sigma_\mathbb{Z} \colon \ldots \forall i_k.\sigma_\mathbb{Z} \colon \Big(\varphi_I(i_1, \ldots, i_k) \to \varphi_V(i_1, \ldots, i_k)\Big).$$

[8] In the original work by Bradley, Manna, and Sipma [BMS06], the Array Property Fragment allows an arbitrary number of element logics. In this thesis, however, we are only interested in the case of a single element logic.

The subformula φ_I is called the *index guard*, whereas the subformula φ_V is called the *value constraint*. These subformulas are constrained as follows.

An index guard has to be constructed according to the following syntax:

- An *index base term* has sort $\sigma_\mathbb{Z}$ and is either an integer constant $c \in \mathbb{Z}$ or of the form $c \cdot x$ where $x \in X \setminus \{i_1, \ldots, i_k\}$ is a variable of sort $\sigma_\mathbb{Z}$. Moreover, if t_1 and t_2 are two index base terms, so is $t_1 + t_2$.
- An *index term* has sort $\sigma_\mathbb{Z}$ and is either a variable $i_1, \ldots, i_k$ or an index base term.
- If t_1 and t_2 are two index terms, then $t_1 \leq t_2$ and $t_1 = t_2$ are index guards. In addition, if φ_1 and φ_2 are two index guards, so are $\varphi_1 \land \varphi_2$ and $\varphi_1 \lor \varphi_2$.

To ease notation, we add syntactic sugar: for two variables $x, x' \in X$ with $x \notin \{i_1, \ldots, i_k\}$ or $x' \notin \{i_1, \ldots, i_k\}$, we allow index guards of the form $x < x'$, $\neg(x \leq x')$, and $\neg(x < x')$; the first can be expressed by $x + 1 \leq x'$ if $x \notin \{i_1, \ldots, i_k\}$ and by $x < x' - 1$ if $x' \notin \{i_1, \ldots, i_k\}$, the second by $x' < x$, and the third by $x' \leq x$. Note, however, that such index guards cannot be constructed from two index variables $i, i' \in \{i_1, \ldots, i_k\}$ since arithmetic on index variables is disallowed.

A value constraint has to be a quantifier-free first-order Σ_{APF}-formula that satisfies the following two conditions:

- The variables $i_1, \ldots, i_k$ must only occur in array read operations of the form $a[i]$.
- Nested read operations such as $a_1[a_2[i]]$ are not allowed.

Finally, the Array Property Fragment consists of all existentially-closed Boolean combinations of array properties and quantifier-free Σ_{APF}-formulas. Let us illustrate its usage with an example.

Example 2.8. Let us consider arrays containing values that can be totally ordered. Moreover, let us assume that an element logic with signature $\Sigma_E = (\{\sigma_E\}, \emptyset, \{<_E, =_E\})$ is given.

A common property that one would like to express is the sortedness of arrays. We can express this property with the following formula where a is an array variable of sort σ_A:

$$\varphi_1(a) := \forall i.\sigma_\mathbb{Z} \colon \forall j.\sigma_\mathbb{Z} \colon (i \leq j) \rightarrow (a[i] <_E a[j] \lor a[i] =_E a[j]).$$

A second common property is the equality of two arrays, which can be expressed by the formula

$$\varphi_2(a_1, a_2) := \forall i.\sigma_\mathbb{Z} \colon a_1[i] =_E a_2[i],$$

where a_1 and a_2 are two array variables of sort σ_A. Note that the Array Property Fragment does not allow to express the equality of two arrays by formulas of the form $a_1 = a_2$. ◁

Most today's SMT solvers implement a decision procedure for the Array Property Fragment. The underlying idea is to treat arrays as uninterpreted functions (where the array index is the only function argument) and to replace read and write operations by function applications. Many SMT solvers (e.g., Microsoft's Z3) also provide a model if a formula is satisfiable. The complexity of such a decision procedure clearly depends on the complexity of the decision procedure used for the underlying element logic. We refer the reader to Kroening and Strichman [KS08] for more details about decision procedures for the Array Property Fragment.

Strand

The logic *Strand (structure and data logic)*, proposed by Madhusudan, Parlato, and Qiu [MPQ11], is the analogue of the Array Property Fragment for recursive heap data structures, such as lists and trees. Strand allows expressing constraints both on the heap structure and on the data contained.

Unlike the logics discussed so far, Strand is a *monadic second-order logic*, which allows quantifying over sets of heap locations. As such, its formal definition is more complex than that of a first-order logic. Since we are mainly interested in using Strand, we refer the reader to Madhusudan, Parlato, and Qiu [MPQ11] for a complete definition and rather sketch the gist of Strand here.

Similar to the Array Property Fragment, Strand combines a first-order element logic over a single sort with a monadic second-order *location logic*, which is used to express conditions on heap locations. A Strand formula is of the form

$$\exists x_1 \colon \ldots \exists x_m \colon \forall y_1 \colon \ldots \forall y_n \colon \varphi(x_1, \ldots, x_m, y_1, \ldots, y_n),$$

where $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are location variables and φ is a monadic second-order formula constructed from atomic formulas of two types, denoted by α and γ. A formula of type α is an atomic formula of the location logic. A formula of type γ is an atomic formula of the element logic in which the data referenced by the location variables $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ can be dereferenced using the expressions $d(x_i)$ and $d(y_j)$; the data at other (free or quantified) location variables cannot be dereferenced.

A model of a Strand formula is a recursive data structure, represented as a triple $(\mathfrak{M}_L, \mathfrak{M}_D, M)$, that satisfies the formula. $\mathfrak{M}_L$ is a model of the location logic, which represents a labeled graph with vertex set M_L, and $\mathfrak{M}_D$ is model for the data logic

with M_D as the underlying domain. Finally, $M\colon M_L \to M_D$ is an interpretation of the function d that maps heap locations to the data they contain.

To illustrate the usage of Strand, let us reconsider the array properties of Example 2.8.

Example 2.9. The first property of Example 2.8 is the sortedness of an array. Expressing the sortedness of a list, which is referenced by the pointer variables *head* and *tail*, can be done using the Strand formula

$$\forall y_1 : \forall y_2 : \Big((head \to^* y_1 \land y_1 \to^* y_2 \land y_2 \to^* tail) \to (d(y_1) \leq d(y_2)) \Big).$$

Thereby, the binary predicate $\to^*$ is part of the location logic and interpreted as the reachability relation; moreover, we assume that $\leq$ is interpreted as the less-than relation over the data domain.

The second property of Example 2.8 is the equality of two arrays. The equality of two lists, however, cannot be expressed in Strand. In fact, it is not even possible to express that two lists have equal length (provided that the location logic does not contain a corresponding predicate). Since the properties we consider later are always expressible in Strand, the expressive power of Strand is of no direct concern to us and, hence, a proof of these claims is out of the scope of this thesis. ◁

Madhusudan, Parlato, and Qiu [MPQ11] noted that the satisfiability problem for Strand is undecidable in general. For this reason, they have introduced a *decidable syntactic fragment* of Strand. The defining characteristics of this fragment are the following two restrictions:

- The subformula $\varphi(x_1,\ldots,x_m,y_1,\ldots,y_n)$ must not contain any further quantification.
- In $\varphi(x_1,\ldots,x_m,y_1,\ldots,y_n)$, the universally quantified variables $y_1,\ldots,y_n$ can only be related by so-called *elastic relations* (whereas the existentially quantified variables $x_1,\ldots,x_m$ can be related by any kind of relation). Intuitively, *elasticity* captures the notion that it cannot be checked whether the variables $y_1,\ldots,y_n$ reference cells that are a bounded distance away. (The relation $\to^*$ in Example 2.9 is an example of an elastic relation.)

Madhusudan, Parlato, and Qiu [MPQ11] described a decision procedure for the decidable syntactic fragment of Strand, which was later improved [MQ11]. However, this procedure has not yet been implemented as an integrated tool and currently requires the user to run several tools manually. The authors demonstrated the applicability of their decision procedure on various examples but did not provide an estimation of the time and space complexity of their algorithm.

2.3 Infinite Games

This section introduces basic definitions and notations of infinite-duration two-player games over graphs. Later, in Chapters 6 and 7, we move away from this basic setting and consider more sophisticated games; we make the necessary changes to the basic definitions in the corresponding chapters. The definitions in this section follow the textbook by Grädel, Thomas, and Wilke [GTW02].

Arenas and Plays

A game consists of an arena and a winning condition. We first introduce arenas and later add winning conditions.

An *arena* is a triple $\mathfrak{A} = (V_0, V_1, E)$ composed of two disjoint countable sets V_0, V_1 of vertices and a directed edge-relation $E \subseteq (V_0 \cup V_1) \times (V_0 \cup V_1)$. The set of all vertices, which we denote by V, is the union of V_0 and V_1; with this notation, the edge relation can be written more concisely as $E \subseteq V \times V$. If V_0 and V_1 are finite, we call $\mathfrak{A}$ a *finite arena*; otherwise, we call $\mathfrak{A}$ an *infinite arena*. To avoid dealing with plays that end prematurely, we assume that an arena does not contains dead ends (i.e., for all vertices $v \in V$ there exists a successor $v' \in V$ with $(v, v') \in E$). This is no restriction since a "sink-vertex" can be added if necessary.

A game is played by two players called *Player 0* and *Player 1*. We often fix a player $\sigma \in \{0, 1\}$, usually Player 0, and refer to the other player as Player σ's opponent or simply as Player $1 - \sigma$. For the sake of simplicity, we use the generic-singular pronoun "he" for both players.

Before we continue with our definitions, it is helpful to picture how a game is played. A play is started by placing a token on some vertex $v \in V$. Then, the player who has control over this vertex moves the token along an edge $(v, v') \in E$ to the successor vertex v'; that is, if $v \in V_0$, then Player 0 moves the token and, conversely, if $v \in V_1$, then Player 1 moves the token. Both players repeat this process ad infinitum, thereby producing an infinite sequence of vertices, which we call a *play*. Finally, we determine the winner of the play based on the vertices occurring during this play.

Formally, a *play* is an infinite sequence $\pi = v_0 v_1 v_2 \ldots \in V^\omega$ of vertices $v_i \in V$ such that $(v_i, v_{i+1}) \in E$ is satisfied for all $i \in \mathbb{N}$; remember that V^ω denotes the set of all infinite sequences of elements from V. A *finite play* is defined analogously as a finite sequence $\rho = v_0 \ldots v_n \in V^*$ of vertices where $(v_i, v_{i+1}) \in E$ holds for all $i \in [n]$. Note that every finite prefix of a play is a finite play and that every finite play can be continued to a play (as arenas do not have dead ends). Finally, given a play $\pi = v_0 v_1 \ldots \in V^\omega$, let

$$\mathrm{Occ}(\pi) = \{v \in V \mid \exists i \in \mathbb{N} \colon v = v_i\}$$

denote the set of all vertices *occurring in* π and

$$\mathrm{Inf}(\pi) = \{v \in V \mid \forall i \in \mathbb{N} \colon \exists j \in \mathbb{N} \colon i < j \text{ and } v = v_j\}$$

denote the set of all vertices *occurring infinitely often in* π.

Games and Winning Conditions

A *game* is a tuple $\mathfrak{G} = (\mathfrak{A}, \mathit{Win})$ consisting of an arena $\mathfrak{A}$ with vertex set V and a *winning condition* $\mathit{Win} \subseteq V^\omega$. The winning condition is used to determine the winner of a play by describing which plays are winning for Player 0: we declare Player 0 as the winner of a play π if $\pi \in \mathit{Win}$—we then also say that *Player 0 wins* π; conversely, Player 1 wins a play π if Player 0 does not win π.

Due to the nature of winning conditions, a game currently is an infinite object and, hence, not suited as input to an algorithm. Therefore, it is necessary to represent a winning condition differently by means of a finite object. To this end, several different types of winning conditions and, consequently, types of games have been introduced in the literature. Although we are in this thesis mainly interested in reachability and safety games, let us also briefly introduce a few other types of games that we consider later.

Let $\mathfrak{G} = (\mathfrak{A}, \mathit{Win})$ be a game over the arena $\mathfrak{A}$ with vertex set V.

- The game $\mathfrak{G}$ is called a *reachability game* if a set $F \subseteq V$ exists such that

  $$\mathit{Win} = \{\pi \in V^\omega \mid \mathrm{Occ}(\pi) \cap F \neq \emptyset\};$$

  that is, Player 0 wins a play if he can eventually reach a vertex of F.

- The game $\mathfrak{G}$ is called a *safety game* if a set $F \subseteq V$ exists such that

  $$\mathit{Win} = \{\pi \in V^\omega \mid \mathrm{Occ}(\pi) \subseteq F\}.$$

  In other words, Player 0 wins a play if he can manage to visit only vertices of F.

- The game $\mathfrak{G}$ is called a *Büchi game* if a set $F \subseteq V$ exists such that

  $$\mathit{Win} = \{\pi \in V^\omega \mid \mathrm{Inf}(\pi) \cap F \neq \emptyset\}.$$

  In a Büchi game, Player 0 wins a play if he visits some vertex $v \in F$ infinitely often.

- The game $\mathfrak{G}$ is called a *(min) parity game* if a parity function $c\colon V \to [k]$ for some $k \in \mathbb{N}_+$ exists such that

  $$Win = \{\pi \in V^\omega \mid \min(c(\mathrm{Inf}(\pi))) \text{ is even}\},$$

  where c is lifted to sets by $c(X) = \{c(v) \mid v \in X\}$. In simple words, the parity condition requires that the minimal parity seen infinitely often during a play is even.

- The game $\mathfrak{G}$ is called a *Muller game* if a set $\mathcal{F} \subseteq 2^V$ exists such that

  $$Win = \{\pi \in V^\omega \mid \mathrm{Inf}(\pi) \in \mathcal{F}\}.$$

  In order to win a play in a Muller game, Player 0 has to visit exactly the vertices of a set $F \in \mathcal{F}$ from some point onwards. Muller winning conditions are often referred to as *ω-regular* winning conditions.

For ease of notation, we use a more natural notation in which we substitute *Win* by the finite object that defines this set. More precisely, we denote reachability games, safety games, and Büchi games by $(\mathfrak{A}, F)$, parity games by $(\mathfrak{A}, c)$, and Muller games by $(\mathfrak{A}, \mathcal{F})$. We always make sure that the type of winning condition is clear from the context.

Example 2.10. Figure 2.5 presents our graphical notation of reachability and safety games. We draw Player 0 vertices as circles ◯ and Player 1 vertices as rectangles □. Furthermore, we shade vertices from F gray.

The game depicted in Figure 2.5, let us call it $\mathfrak{G}^\star = (\mathfrak{A}^\star, F^\star)$, consists of the arena $\mathfrak{A}^\star = (V_0^\star, V_1^\star, E^\star)$ with vertex sets $V_0^\star = \{0, 2, 4\}$, $V_1^\star = \{1, 3, 5\}$ and the set $F^\star = \{1, 2\}$. It can be understood as either a reachability or a safety game. Note, however, that our graphical representation does not contain the information whether the game is a reachability or safety game (but we make sure that this information is always clear from the context). ◃

Strategies

In order to define the winner of a game—in contrast to the winner of a play—, we need to introduce the concept of strategies. Roughly speaking, a strategy tells a player how to continue a play depending on the finite prefix of the play played so far. A strategy is called winning from some vertex $v \in V$ if the player wins every play starting in v that is played according to this strategy.

To define strategies, let us fix a game $\mathfrak{G} = (\mathfrak{A}, Win)$ with vertex set V. A *strategy* for Player σ, $\sigma \in \{0, 1\}$, is a function $f_\sigma\colon V^* V_\sigma \to V$ that satisfies $(v_n, f_\sigma(v_0 \ldots v_n)) \in E$ for

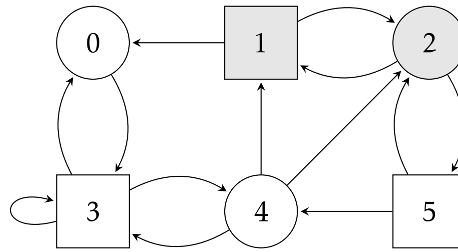


Figure 2.5: The game $\mathfrak{G}^\star$ discussed in Example 2.10.

all $v_0 \dots v_n \in V^* V_\sigma$ (i.e., the next move determined by the strategy respects the edge relation). A play $\pi = v_0 v_1 \dots \in V^\omega$ is *consistent* with a strategy f_σ—or played according to f_σ—if $v_{n+1} = f_\sigma(v_0 \dots v_n)$ holds for every $n \in \mathbb{N}$ with $v_n \in V_\sigma$. Similarly, a finite play $\rho = v_0 \dots v_m \in V^*$ is consistent with f_σ if $v_{n+1} = f_\sigma(v_0 \dots v_n)$ holds for every $n < m$ with $v_n \in V_\sigma$. Finally, we call a strategy f_σ *positional* if the next move $f_\sigma(v_0 \dots v_n)$ exclusively depends on the current vertex v_n (i.e., $f_\sigma(uv) = f_\sigma(wv)$ holds for all $u, w \in V^*$ and $v \in V_\sigma$); in this case, we use the more compact notation $f_\sigma \colon V_\sigma \to V$.

A strategy f_σ is said to be a *winning strategy* for Player σ in a game $\mathfrak{G}$ from a set $U \subseteq V$ of vertices if Player σ wins all plays that start in a vertex $v \in U$ and that are played according to f_σ. For a vertex $v \in U$, we also say that f_σ is a winning strategy for Player σ from v.

The set of vertices from which Player σ has a winning strategy is called the *winning region* of Player σ, denoted by W_σ. If a strategy f_σ is winning from every vertex of the winning region, we call f_σ a *uniform winning strategy*. Due to the definition of winning strategies, it is not hard to verify that $W_0 \cap W_1 = \emptyset$ holds true for every game. On the other hand, we call a game *determined* if $W_0 \cup W_1 = V$ is satisfied. It is well-known that all games presented above are determined.

Theorem 2.2 ([GTW02])**.** *Reachability, safety, Büchi, parity, and Muller games are determined.*

The term *solving a game* denotes the process of determining the winning regions and computing a winning strategy for at least one of the players, typically for Player 0. Since we consider determined games only, it is indeed enough to compute the winning region W_σ of one of the players. The winning region of the opposing player then is $W_{1-\sigma} = V \setminus W_\sigma$.

In the next section, we briefly recap how to solve reachability and safety games.

Solving Reachability and Safety Games

The allegedly simplest way to solve reachability and safety games is based on so-called *attractors* and *attractor strategies*.

Given a game over an arena $\mathfrak{A} = (V_0, V_1, E)$, the *attactor of Player σ with respect to a set $X \subseteq V$*, denoted by $\mathrm{Attr}_\sigma(X)$, consists of all vertices from which Player σ can force a play to eventually reach a vertex in X. Formally, one defines $\mathrm{Attr}_\sigma(X)$ inductively by

$$\mathrm{Attr}^0_\sigma(X) = X$$

and

$$\mathrm{Attr}^{i+1}_\sigma(X) = \mathrm{Attr}^i_\sigma(X) \cup \mathrm{Pre}_\sigma(\mathrm{Attr}^i_\sigma(X)),$$

where $i \in \mathbb{N}_+$ and $\mathrm{Pre}_\sigma(Y)$ is defined for a set $Y \subseteq V$ by

$$\begin{aligned}\mathrm{Pre}_\sigma(Y) = \{&v \in V_\sigma \mid \exists v' \in Y : (v, v') \in E\} \cup \\ &\{v \in V_{1-\sigma} \mid \forall v' \in V : (v, v') \in E \rightarrow v' \in Y\};\end{aligned}$$

the set $\mathrm{Pre}_\sigma(Y)$ contains all vertices from which Player σ can move to a vertex in Y and from which Player $1-\sigma$ has no choice but to move to a vertex in Y. With that defined, the attractor of Player σ with respect to X is the union

$$\mathrm{Attr}_\sigma(X) = \bigcup_{i \in \mathbb{N}} \mathrm{Attr}^i_\sigma(X).$$

We often refer to $\mathrm{Attr}_\sigma(X)$ simply as the *attractor* if σ and X are clear from the context.

Let us first describe how to solve reachability games. A standard induction over the length of play prefixes shows that Player σ can indeed force a play starting in $\mathrm{Attr}_\sigma(X)$ to eventually visit a vertex in X. The winning region of Player 0 in the reachability game $\mathfrak{G} = (\mathfrak{A}, F)$ is, therefore, $W_0 = \mathrm{Attr}_0(F)$. Consequently, the winning region of Player 1 is $W_1 = V \setminus \mathrm{Attr}_0(F)$ as reachability games are determined.

A winning strategy for Player 0 is to decrease the distance to the set F in each move until the play reaches F (provided the play started in W_0). In fact, one can define a positional winning strategy $f_0 \colon V_0 \rightarrow V$ for Player 0 as follows:

- If $v \in W_0 \setminus F$, let $i \in \mathbb{N}$ be minimal such that $v \in \mathrm{Attr}^i_0(F)$ and $v \notin \mathrm{Attr}^{i-1}_0(F)$. Then, $f_0(v) = v'$ for a vertex $v' \in \mathrm{Attr}^{i-1}_0(F)$ with $(v, v') \in E$. The definition of $\mathrm{Attr}_0(F)$ guarantees that the vertex v' exists.

- If $v \in F$ or $v \in W_1$, then $f_0(v) = v'$ for an arbitrary $v \in V$ with $(v, v') \in E$.

Conversely, a winning strategy for Player 1 in a reachability game is to stay in W_1. One obtains a corresponding positional winning strategy $f_1 \colon V_1 \rightarrow V$ for Player 1 as follows:

- If $v \in W_1$, then $f_1(v) = v'$ for an arbitrary $v \in W_1$ with $(v,v') \in E$. Also here, the definition of $\mathrm{Attr}_0(F)$ guarantees that the vertex v' exists.

- If $v \in W_0$, then $f_1(v) = v'$ for an arbitrary $v \in V$ with $(v,v') \in E$.

In reachability and safety games, the exact moves in vertices from which a player cannot win or already has won are irrelevant. Therefore, we usually omit to specify such moves when defining positional winning strategies.

To solve a safety game $\mathfrak{G} = (\mathfrak{A}, F)$, it is in fact sufficient to solve the reachability game $\mathfrak{G}' = (\mathfrak{A}, V \setminus F)$ in which the roles of both player are swapped: the winning region W_σ in $\mathfrak{G}'$ coincides with the winning region $W_{1-\sigma}$ in $\mathfrak{G}$, and a winning strategy for Player σ in $\mathfrak{G}'$ is a winning strategy for Player $1-\sigma$ in $\mathfrak{G}$.

For a reachability or safety game over a finite arena, the inductive definition of the attractor immediately provides an algorithm to solve the game. The key insight is that the attractor is *monotonic* (i.e., $\mathrm{Attr}_\sigma^i(X) \subseteq \mathrm{Attr}_\sigma^{i+1}(X)$ holds for all $X \subseteq V$ and $i \in \mathbb{N}$). Thus, a fixed point exists, and the attractor is the set

$$\mathrm{Attr}_\sigma(X) = \bigcup_{i=0}^{|V|} \mathrm{Attr}_\sigma^i(X) = \mathrm{Attr}_\sigma^{|V|}(X).$$

To compute the attractor, a straightforward fixed-point algorithm suffices: starting with the set F, one collects further vertices according to the inductive definition of the attractor until a fixed point is reached. An efficient implementation of this procedure that runs in time $\mathcal{O}(|E|)$ exists and yields a linear time algorithm to solve reachability and safety games. Note, however, that fixed-point algorithms that build upon the inductive definition of the attractor not necessarily converge if the arena is infinite.

We conclude the section on games with an example that illustrates the attractor computation.

Example 2.11. We continue Example 2.10 (see Figure 2.5 on Page 37) and show how to solve the game $\mathfrak{G}^\star$ assuming that $\mathfrak{G}^\star$ is a reachability game.

We first compute the sets $\mathrm{Attr}_0^i(F)$ step-by-step starting with $\mathrm{Attr}_0^0(F)$ until this process becomes stationary. The resulting set is then $\mathrm{Attr}_0(F)$. In our example, we obtain the following:

$\mathrm{Attr}_0^0(F) = F = \{1,2\}$;
$\mathrm{Attr}_0^1(F) = \{1,2,4\}$; and
$\mathrm{Attr}_0^2(F) = \{1,2,4,5\} = \mathrm{Attr}_0^3(F) = \mathrm{Attr}_0^4(F) = \dots$

Hence, $W_0 = \{1, 2, 4, 5\}$ and $W_1 = V \setminus W_0 = \{0, 3\}$. Moreover,

$$f_0(v) = \begin{cases} 3 & \text{if } v = 0; \\ 1 & \text{if } v = 2; \\ 1 & \text{if } v = 4; \end{cases}$$

is a winning strategy for Player 0 whereas

$$f_1(v) = \begin{cases} 0 & \text{if } v = 1; \\ 3 & \text{if } v = 3; \\ 4 & \text{if } v = 5; \end{cases}$$

is a winning strategy for Player 1. ◃

3 Algorithmic Learning of Finite Automata

Algorithmic learning of finite automata—often simply termed *automata learning*—refers to a variety of techniques that construct finite automata—in most cases DFAs—from externally provided information. One usually groups these techniques by different criteria, most commonly by what type of automaton they learn and whether or not they are allowed to actively query for information. Table 3.1 gives a by far not complete overview of popular learning algorithms. We apply several of these algorithms in the course of this thesis.

Traditionally, the literature on automata learning distinguishes between two different settings: *passive* and *active* learning.[1]

[1] Probabilistic learning, such as *PAC learning* [Val84, KV94], is also common but not subject of this thesis.

Table 3.1: An overview of popular learning algorithms grouped by their target concept (i.e., whether they learn DFAs or NFAs) and their mode of operation (i.e., whether they work in an active or passive learning setting).

	Active learning	Passive learning
DFA	Angluin [Ang87] Kearns and Vazirani [KV94] Rivest and Schapire [RS93]	Biermann and Feldman [BF72] Grinchtein, Leucker, and Piterman [GLP06] Heule and Verwer [HV10] RPNI [OG92]
NFA	NL* [BHKL09]	DeLeTe2 [DLT04] nondeterministic RPNI [OG92]

In a passive learning setting, a learning algorithm is confronted with a finite set of words that have some kind of classification attached; for brevity, we call such words *classified words*. The passive learning task is to compute a finite automaton that is *consistent* with the given words (i.e., that classifies the words correctly). Moreover, the learning algorithm is often required to produce a smallest consistent automaton with respect to the number of states (which is not necessarily unique). A common setting (e.g., studied by Gold [Gol78], Biermann and Feldman [BF72], Trakhtenbrot and Barzdin [TB73], and others) is one in which the words are classified as either *positive* ("has to be accepted") or *negative* ("has to be rejected"), and the task is to compute a minimal consistent DFA.

In an active learning setting, on the other hand, the task of a learning algorithm—often called the *learner*—is to learn a regular language in interaction with a *teacher*. This teacher has knowledge about the language in question and answers queries posed by the learner. The learner is usually allowed to ask *membership* and *equivalence* queries. The fist type of queries asks for the classification of a word with respect to the language in question. The latter type asks whether a conjectured automaton is equivalent to the language the teacher has in mind. The learning stops once the teacher answers an equivalence query positively.

Both passive and active learning have become generic concepts, which can be and have been adapted and extended far beyond their original scope. Hence, we need to carefully formalize how our learning settings look like. To be able to do so, we first introduce both the original passive learning setting (in Section 3.1) and the original active learning setting (in Section 3.2); whenever changes to these settings are necessary in the course of this thesis, we make them in the respective chapters. Sections 3.1 and 3.2 also serve as a detailed reference to the learning algorithms that we apply or modify later in this thesis. The present chapter concludes in Section 3.3 with a presentation of the Libalf automata learning library, which we used extensively for our implementations and experiments.

3.1 Passive Learning of DFAs

In the original passive learning setting (e.g., studied by Biermann and Feldman [BF72]), a learning algorithm is confronted with a finite number of classified words, which are bundled in a so-called *sample*. Such a sample is a pair $\mathcal{S} = (S_+, S_-)$ consisting of two disjoint, finite sets $S_-, S_+ \subseteq \Sigma^*$ of words over a common alphabet Σ. Intuitively, the set S_+ contains *positively classified words* that an automaton has to accept whereas the set S_- contains *negatively classified* words that the automaton has to reject. Having this in mind, we call a DFA (or an NFA for that matter) $\mathcal{A}$ *consistent with* $\mathcal{S}$ if it accepts all

words in S_+ and rejects all words in S_- (i.e., if it satisfies $S_+ \subseteq L(\mathcal{A})$ and $S_- \cap L(\mathcal{A}) = \emptyset$). If $\mathcal{S}$ is clear from the context, we also say that $\mathcal{A}$ is a consistent DFA (respectively consistent NFA).

Using these notations, we can now formulate the passive learning setting we are interested in.

Definition 3.1 (Passive learning of DFAs)**.** Given a sample $\mathcal{S} = (S_+, S_-)$, the *passive learning task* is to compute a DFA $\mathcal{A}$ of minimal size that is consistent with $\mathcal{S}$ (i.e., that satisfies $S_+ \subseteq L(\mathcal{A})$ and $S_- \cap L(\mathcal{A}) = \emptyset$).

Let us first have a closer look at a relaxed version of passive learning without the minimality constraint. In this case, the learning task is easy, and the so-called *prefix acceptor* is already a solution. Broadly speaking, the prefix acceptor is a DFA that keeps track of the input read so far as long as the input is a prefix of a word in S_+. If the input does not belong to $\mathrm{Pref}(S_+)$, the DFA moves to a special sink-state $\bot$ from which all continuing runs are rejecting. Formally, the prefix acceptor for a sample $\mathcal{S} = (S_+, S_-)$ is the DFA $\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}} = (\mathrm{Pref}(S_+) \cup \{\bot\}, \Sigma, \varepsilon, \delta, S_+)$ with

$$\delta(q, a) = \begin{cases} ua & \text{if } q = u \text{ and } ua \in \mathrm{Pref}(S_+); \\ \bot & \text{otherwise;} \end{cases}$$

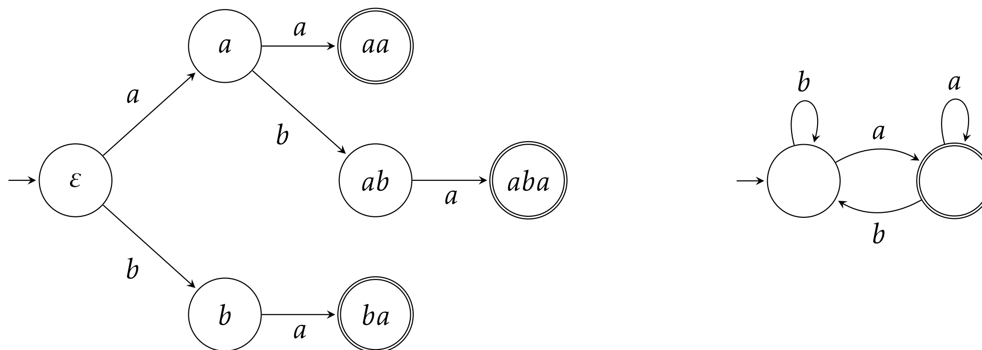
where $q \in \mathrm{Pref}(S_+) \cup \{\bot\}$ and $a \in \Sigma$. A straightforward induction over the length of input-words shows that $\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}}$ accepts S_+. Therefore, $\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}}$ is a solution of the passive learning task without the minimality constraint because $\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}}$ satisfies both $S_+ \subseteq L(\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}})$ and $S_- \cap L(\mathcal{A}^{\mathcal{S}}_{\mathrm{Pref}}) = \emptyset$.

Algorithms such as the *regular positive negative inference* algorithm *RPNI* [OG92] and the *Blue-fringe* algorithm [LPP98] try to reduce the size of the prefix acceptor further by merging its states as long as the resulting automaton stays consistent with the sample. These heuristics run in time polynomial in the combined length of all words in the sample and often produce results smaller than the prefix acceptor, but cannot guarantee to find a consistent DFA of minimal size.

However, if the learner has to come up with a minimal consistent DFA, passive learning becomes hard. In fact, Gold [Gol78] showed that the corresponding decision problem

> "Given a sample $\mathcal{S}$ and a natural number $k \in \mathbb{N}$. Does a DFA with k states that is consistent with $\mathcal{S}$ exist?"

is NP-complete using a reduction from the satisfiability problem of formulas in Propositional Boolean Logic. The intuitive reason is that the behavior of a consistent DFA on words not belonging to S_+ and S_- can be arbitrary. As a consequence, a learner



(a) The prefix acceptor $\mathcal{A}^{\mathcal{S}}_{\text{Pref}}$. The sink-state and all transitions leading there are omitted.

(b) A minimal DFA that is consistent with $\mathcal{S}$.

Figure 3.1: The prefix acceptor $\mathcal{A}^{\mathcal{S}}_{\text{Pref}}$ for the sample $\mathcal{S} = (S_+, S_-)$ where $S_+ = \{aa, ba, aba\}$ and $S_- = \{\varepsilon, ab\}$ and a minimal consistent DFA.

needs to choose the behavior of the resulting DFA on those words carefully in order to obtain a DFA of minimal size.

The following example illustrates the passive learning task: on the one hand, strictly according to Definition 3.1 and, one the other hand, without the minimality constraint.

Example 3.1. Consider the sample $\mathcal{S} = (S_+, S_-)$ with

$$S_+ = \{aa, ba, aba\} \text{ and } S_- = \{\varepsilon, ab\}.$$

Figure 3.1a shows the prefix acceptor $\mathcal{A}^{\mathcal{S}}_{\text{Pref}}$. The sink-state $\perp$ and all transitions leading there are omitted for sake of readability.

A minimal DFA that is consistent with $\mathcal{S}$ is depicted in Figure 3.1b. It is not hard to verify that this DFA is indeed consistent with $\mathcal{S}$. Moreover, it is a minimal consistent DFA since every DFA with just one state accepts either all words or none.

Finally, let us note that the prefix acceptor can be arbitrary larger than a minimal consistent DFA. A simple case in which this phenomenon occurs is the family of samples $\mathcal{S}_n = (\{a^n\}, \emptyset)$ for $n \in \mathbb{N}$. The prefix acceptor $\mathcal{A}^{\mathcal{S}_n}_{\text{Pref}}$ for this sample has $n + 2$ states whereas the—this time unique—minimal consistent DFA has always only one state. ◃

Despite the hardness of passive learning, several learning algorithms have been proposed. Since we make extensive use of passive learning techniques in Chapter 4, the reader might find an introduction to the most significant algorithms helpful. In Sections 3.1.1 to 3.1.5, we describe Biermann and Feldman's method [BF72], Grinchtein, Leucker, and Piterman's method [GLP06], Heule and Verwer's methods [HV10], as well as a novel approach based on first-order logic with uninterpreted functions [NJ13].

All of these algorithms are based on translations of the passive learning task into a satisfiability problem of logic formulas. The rationale behind these approaches is that passive learning is computationally hard and logic solvers offer effective and reasonably efficient means for finding solutions.

In Section 3.1.6, we conclude with a thorough evaluation of the methods mentioned above. This evaluation includes a theoretical comparison of the resulting logic formulas with respect to their size, the number of variables, and so on. Moreover, we experimentally benchmark implementations of these methods in order to get a better understanding of their practical applicability (which clearly depends on the logic formulas and the underlying logic solvers).

Parts of this section, though without the benchmark, appeared in [LN12].

3.1.1 Biermann and Feldman's Method

Biermann and Feldman [BF72] were among the first to study the passive learning task. Their method is also the basis of all other techniques described here and works as described next.

Given a sample $\mathcal{S} = (S_+, S_-)$ over an alphabet Σ, Biermann and Feldman's idea is to consider the runs of a (minimal) consistent DFA on all prefixes of words in $S_+ \cup S_-$ in a manner similar to the prefix acceptor. To this end, they denote the state that the DFA reaches after reading a word $u \in \mathrm{Pref}(S_+ \cup S_-)$ by x_u. Since the DFA is unknown, Biermann and Feldman think of each x_u as a variable (ranging over the states) and impose constraints that enforce these variables to encode runs of a DFA on the words from the sample. This leads them to the following formulas in Equality Logic:

$$\bigwedge_{ua, u'a \in \mathrm{Pref}(S_+ \cup S_-)} x_u = x_{u'} \rightarrow x_{ua} = x_{u'a} \tag{3.1}$$

$$\bigwedge_{u \in S_+, u' \in S_-} x_u \neq x_{u'} \tag{3.2}$$

The first formula ensures that whenever the prospective DFA reaches the same state after reading the inputs u and u', then it must also reach the same state after reading ua and $u'a$. The second formula enforces that the run on two differently classified words never ends in the same state.

Let $\varphi(\overline{x})$ be the conjunction of Formulas 3.1 and 3.2 where $\overline{x}$ is the list of the variables x_u with $u \in \mathrm{Pref}(S_+ \cup S_-)$. Since specific names for the states are unimportant, a suitable domain for the variables is the set of natural numbers. Moreover, since $\varphi(\overline{x})$ consists solely of Boolean combinations of equality and inequality constraints, the actual values of the variables are unimportant as long as the constraints are satisfied.

Thus, we can assume without loss of generality that the values of variables in a model range over the set $[n]$ for a suitable $n \in \mathbb{N}_+$.

Given a model $\mathfrak{M}$ of $\varphi(\overline{x})$ whose variables range over the set $[n]$, we derive the DFA encoded by $\mathfrak{M}$, which we denote by $\mathcal{A}_{\mathfrak{M}}$, as described next.

Definition 3.2. Let $\mathfrak{M}$ be a model of $\varphi(\overline{x})$ whose variables range over the set $[n]$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0^{\mathfrak{M}}, \delta, F)$ is defined by

- $Q = \{q_0, \ldots, q_{n-1}\}$;
- $q_0^{\mathfrak{M}} = q_i$ if and only if $x_\varepsilon^{\mathfrak{M}} = i$;
- $\delta(q_i, a) = \begin{cases} q_j & \text{if } ua \in \mathrm{Pref}(S_+ \cup S_-) \text{ exists such that } x_u^{\mathfrak{M}} = i \text{ and } x_{ua}^{\mathfrak{M}} = j; \\ q_i & \text{otherwise;} \end{cases}$
  where $q_i, q_j \in Q$ and $a \in \Sigma$; and
- $F = \{q_i \in Q \mid \exists u \in S^+ \colon x_u^{\mathfrak{M}} = i\}$.

Note that a model of $\varphi(\overline{x})$ does not determine the DFA $\mathcal{A}_{\mathfrak{M}}$ completely as it lacks information about transitions in cases where no $ua \in \mathrm{Pref}(S_+ \cup S_-)$ with $x_u^{\mathfrak{M}} = i$ and $x_{ua}^{\mathfrak{M}} = j$ exists. However, such transitions are unimportant towards the consistency of $\mathcal{A}_{\mathfrak{M}}$ because they are never used when reading words of the sample. Thus, adding self-loops (as we did in Definition 3.2) is just one possibility to define these transitions.

The following lemma states that Definition 3.2 indeed produces a DFA that is consistent with the given sample.

Lemma 3.1. *Let $\mathcal{S}$ be a sample, $\mathfrak{M}$ a model of $\varphi(\overline{x})$ whose variables range over the set $[n]$, and $\mathcal{A}_{\mathfrak{M}}$ as constructed in Definition 3.2. Then, $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$.*

Proof of Lemma 3.1. We split the proof in three parts:[2]

1. We show that the DFA $\mathcal{A}_{\mathfrak{M}}$ is well-defined.
2. We show that $\mathfrak{M}$ in fact encodes runs of $\mathcal{A}_{\mathfrak{M}}$ on words from $\mathrm{Pref}(S_+ \cup S_-)$.
3. We conclude, using the knowledge from Part 2, that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$.

Let us now show the first part of the proof and argue that $\mathcal{A}_{\mathfrak{M}}$ is well-defined. At first, it is not hard to verify that the initial state and the set of final states are well-defined. Furthermore, δ is a well-defined deterministic transition function because Formula 3.1 ensures that the definition of $\delta(q_i, a)$ is independent of the actual choice of the word $ua \in \mathrm{Pref}(S_+ \cup S_-)$ and, thus, of the variables x_u and x_{ua}.

In order to prove the second part, we show by induction over the length of words $u \in \mathrm{Pref}(S_+ \cup S_-)$ that $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u} q_i$ implies $x_u^{\mathfrak{M}} = i$.

[2] We use this proof strategy also for other proofs that follow in this section.

Base case Let $u = \varepsilon$. By definition of runs, we have $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{\varepsilon} q_0^{\mathfrak{M}}$. In addition, $q_0^{\mathfrak{M}} = q_i$ if and only if $x_\varepsilon^{\mathfrak{M}} = i$ holds by definition of $\mathcal{A}_{\mathfrak{M}}$, which proves the claim.

Induction step Let $u = va$ with $va \in \mathrm{Pref}(S_+ \cup S_-)$ and $a \in \Sigma$, and consider the unique run $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{v} q_i \xrightarrow{a} q_j$. Applying the induction hypothesis for v yields $x_v^{\mathfrak{M}} = i$. Moreover, the transition $\delta(q_i, a) = q_j$ is used in the last step of the run of $\mathcal{A}_{\mathfrak{M}}$ on u. Since $u \in \mathrm{Pref}(S_+ \cup S_-)$, we know by definition of $\mathcal{A}_{\mathfrak{M}}$'s transition function that then $x_{va}^{\mathfrak{M}} = x_u^{\mathfrak{M}} = j$ holds.

We can now prove the third part and show that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$. Let us first consider the case $u \in S_+$. Since $\mathcal{A}_{\mathfrak{M}}$ is deterministic, there exists a unique run $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u} q_i$. Thus, the induction above yields $x_u^{\mathfrak{M}} = i$. Moreover, $q_i \in F$ holds by definition of the final states of $\mathcal{A}_{\mathfrak{M}}$ and, hence, $u \in L(\mathcal{A}_{\mathfrak{M}})$. Since $u \in S_+$ was chosen arbitrarily, we obtain $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$. In a similar manner using the definition of $\mathcal{A}_{\mathfrak{M}}$'s final states and Formula 3.2, we deduce $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$. □

Assigning a different value for every variable x_u trivially solves $\varphi(\overline{x})$ (which then yields a DFA similar to the prefix acceptor). Conversely, one can easily construct a model of $\varphi(\overline{x})$ from a consistent DFA by simply looking at the states reached when reading words of the sample. Thus, a model with minimal range yields a minimal consistent DFA. This observation is summarized in the following theorem.

Theorem 3.2 (Biermann and Feldman [BF72]). *Let $\mathcal{S}$ be a sample and $\mathfrak{M}$ a model of $\varphi(\overline{x})$ with minimal range. Then, the DFA $\mathcal{A}_{\mathfrak{M}}$ is a minimal DFA that is consistent with $\mathcal{S}$.*

One can find a model for $\varphi(\overline{x})$ with minimal range in various ways; for instance, Oliveira and Silva [OS01] developed an explicit search algorithm using backtracking, one can use generic CSP solvers, and also most SMT solvers are able to solve CSPs. However, note that finding a model with minimal range involves a search with parameter n of some kind.

Additionally, Grinchtein, Leucker, and Piterman [GLP06] suggest to add further constraints that fix certain variables in order to break symmetries. These constraints are based on the notion of *obviously different words*: two words $u, v \in \mathrm{Pref}(S_+ \cup S_-)$ are *obviously different in $\mathcal{S}$* if there exists a $w \in \Sigma^*$ such that either $uw \in S_+$ and $vw \in S_-$, or $uw \in S_-$ and $vw \in S_+$. It is not hard to see that two obviously different words need to lead to different states in every consistent DFA. Thus, if $\{u_1, \ldots, u_k\} \subseteq \mathrm{Pref}(S_+ \cup S_-)$ is a set of pairwise obviously different words, Grinchtein, Leucker, and Piterman suggest adding the formulas

$$x_{u_i} = i \tag{3.3}$$

for each $i \in \{1, \dots, k\}$. Note, however, that finding a maximal set of pairwise obviously different words itself is NP-complete because it can be reduced to finding a maximum clique in graphs, which is known to be NP-complete. Thus, a heuristic method of some kind should be applied in practice.

The idea of assigning states to obviously different words can easily be applied to all methods described in the following. However, so as to clutter the following presentation not too much with straightforward details, we skip presenting this extension.

3.1.2 Grinchtein, Leucker, and Piterman's Unary SAT Method

With the emerge of efficient SAT solvers, Grinchtein, Leucker, and Piterman [GLP06] proposed to translate Biermann and Feldman's approach into an equivalent satisfiability problem of Propositional Boolean Logic. As we show later, SAT-based approaches can in fact effectively solve reasonably large instances.

Grinchtein, Leucker, and Piterman's key idea is to construct a Boolean formula $\varphi_n^{\mathcal{S}}$ for a natural number $n \geq 1$ that has the following properties:

1. The formula $\varphi_n^{\mathcal{S}}$ is satisfiable if and only if there exists a DFA with n states that is consistent with $\mathcal{S}$.
2. A model of $\varphi_n^{\mathcal{S}}$ contains enough information to derive a consistent DFA.

Although the minimal value of n is not known in advance, one can use a binary search to identify this value. To ease the following description a bit, let us assume that the prospective DFA has the state space $Q = \{q_0, \dots, q_{n-1}\}$.

The formula $\varphi_n^{\mathcal{S}}$ ranges over Boolean variables $x_{u,q}$ where $u \in \mathrm{Pref}(S_+ \cup S_-)$ and $q \in Q$. The meaning of such a variable is that if $x_{u,q}$ is set to *true*, then the prospective DFA reaches the state q after reading the word u. Hence, each variable x_u of Biermann and Feldman's formula φ is encoded in unary by the variables $x_{u,q_0}, \dots, x_{u,q_{n-1}}$ (we also present a binary encoding later). To make this encoding work, one has to make sure that for every $u \in \mathrm{Pref}(S_+ \cup S_-)$ exactly one of the variables $x_{u,q_0}, \dots, x_{u,q_{n-1}}$ is set to *true*. The following two formulas enforce this.

$$\bigwedge_{u \in \mathrm{Pref}(S_+ \cup S_-)} \bigvee_{q \in Q} x_{u,q} \tag{3.4}$$

$$\bigwedge_{u \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{q,q' \in Q, q \neq q'} \neg x_{u,q} \vee \neg x_{u,q'} \tag{3.5}$$

The formulas below are the translation of Biermann and Feldman's constraints into Propositional Boolean Logic; Formula 3.6 corresponds to Formula 3.1, and Formula 3.7

corresponds to Formula 3.2. Formula 3.6 is not shown in conjunctive normal form for the sake of readability.

$$\bigwedge_{ua,u'a \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} (x_{u,p} \wedge x_{u',p}) \rightarrow (x_{ua,q} \leftrightarrow x_{u'a,q}) \tag{3.6}$$

$$\bigwedge_{u \in S_+, u' \in S_-} \bigwedge_{q \in Q} \neg x_{u,q} \vee \neg x_{u',q} \tag{3.7}$$

In order to translate Formula 3.6 into conjunctive normal form, we first split the equivalence $x_{ua,q} \leftrightarrow x_{u'a,q}$ into the disjunction $(x_{ua,q} \wedge x_{u'a,q}) \vee (\neg x_{ua,q} \wedge \neg x_{u'a,q})$. Then, we apply the distributive law to both parts of the disjunction and obtain the following two formulas, which now are in conjunctive normal form.

$$\bigwedge_{ua,u'a \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} \neg x_{u,p} \vee \neg x_{u',p} \vee x_{ua,q} \vee \neg x_{u'a,q} \tag{3.8}$$

$$\bigwedge_{ua,u'a \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} \neg x_{u,p} \vee \neg x_{u',p} \vee \neg x_{ua,q} \vee x_{u'a,q} \tag{3.9}$$

Let $\varphi_n^S(\overline{x})$ be the conjunction of Formulas 3.4 and 3.5 and Formulas 3.7 to 3.9 where $\overline{x}$ is the vector of all variables $x_{u,q}$ for $u \in \mathrm{Pref}(S_+ \cup S_-)$ and $q \in Q$. As before, a model $\mathfrak{M}$ of φ_n^S encodes a DFA, which we also denote by $\mathcal{A}_{\mathfrak{M}}$ for the sake of simplicity; we define this automaton as described next.

Definition 3.3. Let $\mathfrak{M} \models \varphi_n^S(\overline{x})$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0^{\mathfrak{M}}, \delta, F)$ is defined by

- $Q = \{q_0, \ldots, q_{n-1}\}$;
- $q_0^{\mathfrak{M}} = q_i$ if and only if $x_{\varepsilon,q_i}^{\mathfrak{M}} = \mathit{true}$;
- $\delta(q_i, a) = \begin{cases} q_j & \text{if } ua \in \mathrm{Pref}(S_+ \cup S_-) \text{ exists with } x_{u,q_i}^{\mathfrak{M}} = \mathit{true} \text{ and } x_{ua,q_j}^{\mathfrak{M}} = \mathit{true}; \\ q_i & \text{otherwise}; \end{cases}$
  where $q_i, q_j \in Q$ and $a \in \Sigma$; and
- $F = \{q_i \in Q \mid \exists u \in S_+ \colon x_{u,q_i}^{\mathfrak{M}} = \mathit{true}\}$.

Again, note that the exact target-state of the transition $\delta(q_i, a)$ is unimportant in the case that no $ua \in \mathrm{Pref}(S_+ \cup S_-)$ with $x_{u,q_i}^{\mathfrak{M}} = \mathit{true}$ and $x_{ua,q_j}^{\mathfrak{M}} = \mathit{true}$ exists.

The next theorem states that Grinchtein, Leucker, and Piterman's unary method produces a minimal consistent DFA.

Theorem 3.3 (Grinchtein, Leucker, and Piterman [GLP06]). *For a sample $\mathcal{S}$ and $n \in \mathbb{N}$, a model $\mathfrak{M}$ of $\varphi_n^{\mathcal{S}}(\overline{x})$ yields a DFA $\mathcal{A}_{\mathfrak{M}}$ with n states that is consistent with $\mathcal{S}$. A (binary) search can be used to find the minimal n for which $\varphi_n^{\mathcal{S}}(\overline{x})$ is satisfiable, which yields a minimal DFA that is consistent with $\mathcal{S}$.*

Proof of Theorem 3.3. The proof of Theorem 3.3 is similar to the one of Lemma 3.1. We first observe that for every $u \in \mathrm{Pref}(S_+ \cup S_-)$ exactly one variable $x_{u,q}$ is set to *true* due to Formulas 3.4 and 3.5. This observation implies that $\mathcal{A}_{\mathfrak{M}}$ is well-defined: the initial state is well-defined because $x_{\varepsilon,q}^{\mathfrak{M}}$ is *true* for a unique $q \in Q$, and the final states are well-defined due to Formula 3.7; furthermore, Formulas 3.8 and 3.9 ensure that δ is also well-defined.

Once we have established that $\mathcal{A}_{\mathfrak{M}}$ is well-defined, we can use an induction similar to the one in the proof of Lemma 3.1, using Formulas 3.8 and 3.9, to show that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} q$ implies $x_{u,q}^{\mathfrak{M}} = \mathit{true}$ for all $u \in \mathrm{Pref}(S_+ \cup S_-)$. As in the proof of Lemma 3.1, we can use this fact to show that $\mathcal{A}_{\mathfrak{M}}$ is indeed consistent with $\mathcal{S}$.

Finally, by using a binary search, or by increasing the value of n stepwise by one, one can guarantee to find the minimal value of n such that $\varphi_n^{\mathcal{S}}$ is satisfiable. Since one can construct a model of $\varphi_n^{\mathcal{S}}$ from every consistent DFA with n states, a model for the minimal n yields a minimal DFA that is consistent with $\mathcal{S}$. □

3.1.3 Grinchtein, Leucker, and Piterman's Binary SAT Method

Grinchtein, Leucker, and Piterman [GLP06] also proposed an alternative encoding of Biermann and Feldman's formula that represents the values of the variables x_u in binary rather than in unary. In their binary encoding, the state a prospective DFA reaches after reading a word $u \in \mathrm{Pref}(S_+ \cup S_-)$ is encoded by m Boolean variables $x_{u,0}, \ldots, x_{u,m-1}$. To ease the following description, let us assume for the moment that the size n of the prospective DFA is a power of two, say $n = 2^m$ where $m \in \mathbb{N}$.

To translate forth and back between states and their binary encoding in terms of the variables $x_{u,0}, \ldots, x_{u,m-1}$, we introduce two functions bin_m and state_m. The function $\mathrm{bin}_m \colon [2^m] \to \{0,1\}^*$ maps a number $k \in [2^m]$ to its m-bit binary representation in the least-bit-first encoding; for instance, $\mathrm{bin}_4(3) = 1100$. The function state_m, on the other hand, takes a word $u \in \mathrm{Pref}(S_+ \cup S_-)$ and a model $\mathfrak{M}$ as arguments and computes the states encoded by the values of the variables $x_{u,0}$ to $x_{u,m-1}$. More precisely, $\mathrm{state}_m(u, \mathfrak{M}) = q_i$ if and only if $i = \sum_{j=0}^{m-1} 2^{x_{u,j}^{\mathfrak{M}}}$ where we interpret $x_{u,j}^{\mathfrak{M}} = \mathit{false}$ as 0 and $x_{u,j}^{\mathfrak{M}} = \mathit{true}$ as 1.

Towards the translation of Biermann and Feldman's formula into a binary encoding in Propositional Boolean Logic, Grinchtein, Leucker, and Piterman introduce auxiliary formulas $\psi_{u,u'}^{\neq}$ for $u, u' \in \mathrm{Pref}(S_+ \cup S_-)$ that express $x_u \neq x_{u'}$. The formula $\psi_{u,u'}^{\neq}$ is based

on the simple observation that $x_u \neq x_{u'}$ holds if and only if there is a distinguishing bit in their binary representation; in other words, $x_u \neq x_{u'}$ holds if and only if $\bigvee_{k \in [m]} (x_{u,k} \leftrightarrow \neg x_{u',k})$ is satisfied. Hence, when turned into conjunctive normal form, Grinchtein, Leucker, and Piterman obtain the following formula $\psi^{\neq}_{u,u'}(\overline{x})$ where $\overline{x}$ is the list of all variables $x_{u,i}$ for $u \in \mathrm{Pref}(S_+ \cup S_-)$ and $i \in [m]$.

$$
\begin{aligned}
\psi^{\neq}_{u,u'}(\overline{x}) \coloneqq {} & (x_{u,0} \lor x_{u',0} \lor x_{u,1} \lor x_{u',1} \lor \ldots \lor x_{u,m-1} \lor x_{u',m-1}) \land \\
& (x_{u,0} \lor x_{u',0} \lor \neg x_{u,1} \lor \neg x_{u',1} \lor \ldots \lor x_{u,m-1} \lor x_{u',m-1}) \land \\
& (\neg x_{u,0} \lor \neg x_{u',0} \lor \neg x_{u,1} \lor \neg x_{u',1} \lor \ldots \lor x_{u,m-1} \lor x_{u',m-1}) \land \\
& \quad \vdots \\
& (\neg x_{u,0} \lor \neg x_{u',0} \lor \neg x_{u,1} \lor \neg x_{u',1} \lor \ldots \lor \neg x_{u,m-1} \lor \neg x_{u',m-1})
\end{aligned}
$$

Thus, each inequality is expressed by 2^m clauses. However, since m is logarithmic in n, each inequality corresponds to a number of clauses that is linear in the size of the prospective DFA.

To express Formula 3.1 of Biermann and Feldman's encoding, Grinchtein, Leucker, and Piterman exploit that $x_u = x_{u'} \rightarrow x_{ua} = x_{u'a}$ is—by definition—equivalent to $x_u \neq x_{u'} \lor x_{ua} = x_{u'a}$. Given this fact and using the same schema as above, they end up with the following formula.

$$
\bigwedge_{ua,u'a \in \mathrm{Pref}(S_+ \cup S_-)}
\begin{pmatrix}
(\psi^{\neq}_{u,u'}(\overline{x}) \lor x_{ua,0} \lor \neg x_{u'a,0}) \land \\
(\psi^{\neq}_{u,u'}(\overline{x}) \lor \neg x_{ua,0} \lor x_{u'a,0}) \land \\
(\psi^{\neq}_{u,u'}(\overline{x}) \lor x_{ua,1} \lor \neg x_{u'a,1}) \land \\
(\psi^{\neq}_{u,u'}(\overline{x}) \lor \neg x_{ua,1} \lor x_{u'a,1}) \land \\
\vdots \\
(\psi^{\neq}_{u,u'}(\overline{x}) \lor x_{ua,m-1} \lor \neg x_{u'a,m-1}) \land \\
(\psi^{\neq}_{u,u'}(\overline{x}) \lor \neg x_{ua,m-1} \lor x_{u'a,m-1})
\end{pmatrix}
\tag{3.10}
$$

Note that it is not hard to convert Formula 3.10 into conjunctive normal form using the distributive law.

The auxiliary formulas $\psi^{\neq}_{u,u'}$ easily allow translating Formula 3.2 of Biermann and Feldman's encoding, which leads to the formula shown next.

$$
\bigwedge_{u \in S_+, u' \in S_-} \psi^{\neq}_{u,u'}(\overline{x}) \tag{3.11}
$$

Finally, if n is not a power of two, Grinchtein, Leucker, and Piterman add an additional formula that enforces the encoded states to stay inside the set $[n]$. They achieve this by explicitly disallowing all bit representations of values between n and

$2^m - 1$. To this end, they introduce further auxiliary formulas $\psi_u^{m,i}$ that express that the state encoded by $x_{u,0}, \dots, x_{u,m-1}$ is not the state q_i. For $\text{bin}_m(i) = a_0 \dots a_{m-1}$, the formula $\psi_u^{m,i}$ is defined by

$$\psi_u^{m,i} := \bigvee_{j=0}^{m-1} l_{u,j} \text{ where } l_{u,j} = \begin{cases} x_{u,j} & \text{if } a_j = 0; \\ \neg x_{u,j} & \text{if } a_j = 1. \end{cases}$$

Using $\psi_u^{m,i}$, the formula

$$\bigwedge_{u \in \text{Pref}(S_+ \cup S_-)} \bigwedge_{i=n}^{2^m-1} \psi_u^{m,i} \tag{3.12}$$

then restricts the states of a prospective DFA to $Q = \{q_0, \dots, q_{n-1}\}$.

Let $\psi_n^{\mathcal{S}}(\overline{x})$ be the conjunction of Formulas 3.10 and 3.11, as well as Formula 3.12 if n is not a power of two. Again, we can use a model $\mathfrak{M}$ of $\psi_n^{\mathcal{S}}(\overline{x})$ to derive a DFA $\mathcal{A}_{\mathfrak{M}}$ (that is consistent with $\mathcal{S}$).

Definition 3.4. Let $\mathfrak{M} \models \psi_n^{\mathcal{S}}(\overline{x})$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0^{\mathfrak{M}}, \delta, F)$ is defined by

- $Q = \{q_0, \dots, q_{n-1}\}$;
- $q_0^{\mathfrak{M}} = \text{state}_m(\varepsilon, \mathfrak{M})$;
- $\delta(q_i, a) = \begin{cases} q_j & \text{if } ua \in \text{Pref}(S_+ \cup S_-) \text{ exists with } q_i = \text{state}_m(u, \mathfrak{M}) \text{ and} \\ & q_j = \text{state}_m(ua, \mathfrak{M}); \\ q_i & \text{otherwise;} \end{cases}$
  where $q_i, q_j \in Q$ and $a \in \Sigma$; and
- $F = \{q_i \in Q \mid \exists u \in S_+ \colon q_i = \text{state}_m(u, \mathfrak{M})\}$.

Again, note that the exact value of $\delta(q_i, a)$ is unimportant in all cases in which no $ua \in \text{Pref}(S_+ \cup S_-)$ with $q_i = \text{state}_m(u, \mathfrak{M})$ and $q_j = \text{state}_m(ua, \mathfrak{M})$ exists.

The next theorem states that Grinchtein, Leucker, and Piterman's method indeed produces a minimal consistent DFA.

Theorem 3.4 (Grinchtein, Leucker, and Piterman [GLP06])**.** *For a sample $\mathcal{S}$ and $n \in \mathbb{N}$, a model $\mathfrak{M}$ of $\psi_n^{\mathcal{S}}(\overline{x})$ yields a DFA $\mathcal{A}_{\mathfrak{M}}$ with n states that is consistent with $\mathcal{S}$. A (binary) search can be used to find the minimal n for which $\psi_n^{\mathcal{S}}(\overline{x})$ is satisfiable, which yields a minimal DFA that is consistent with $\mathcal{S}$.*

We skip the proof of Theorem 3.4 since it is almost identical to the proof of Theorem 3.3 except for a different encoding of runs.

3.1.4 Heule and Verwer's SAT Method

Heule and Verwer [HV10] suggest a modification of Grinchtein, Leucker, and Piterman's unary encoding, which introduces additional auxiliary variables but typically has less clauses. Heule and Verwer's approach is to encode the prospective DFA directly into the formula using auxiliary variables $d_{p,a,q}$ and f_q where $p,q \in Q$ and $a \in \Sigma$. The meaning is that if $d_{p,a,q}$ is set to *true*, then the prospective DFA contains the transition $\delta(p,a) = q$. Moreover, if f_q is set to *true*, then q is a final state.

To make this encoding work, Heule and Verwer impose constraints on the variables $d_{p,a,q}$, f_q, and $x_{u,q}$ that express two things. First, the variables $d_{p,a,q}$ have to encode a deterministic function. More precisely, Formula 3.13 ensures that for each pair of state p and input-symbol a there exists at most one outgoing transition (thus, the variables $d_{p,a,q}$ encode a deterministic but potentially not total transition function). Second, the variables $x_{u,q}$ have to encode valid runs with respect to the transition function defined by $d_{p,a,q}$. On the one hand, this means that at least one of the variables $x_{u,q_0}, \ldots, x_{u,q_{n-1}}$ is set to *true* for each $u \in \mathrm{Pref}(S_+ \cup S_-)$ (cf. Formula 3.14). On the other hand, $x_{ua,q}$ has to be set to *true* if the prospective DFA reaches state p after reading u and $\delta(p,a) = q$; that is, $x_{ua,q}$ has to be set to *true* if both $x_{u,p}$ and $d_{p,a,q}$ are set to *true* (cf. Formula 3.15). Finally, the prospective DFA has to be consistent with $\mathcal{S}$; that is, words from S_+ have to lead to accepting states while words from S_- have to lead to rejecting states (cf. Formula 3.16).

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{q,q' \in Q, q \neq q'} \neg d_{p,a,q} \vee \neg d_{p,a,q'} \tag{3.13}$$

$$\bigwedge_{u \in \mathrm{Pref}(S_+ \cup S_-)} \bigvee_{q \in Q} x_{u,q} \tag{3.14}$$

$$\bigwedge_{ua \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} (x_{u,p} \wedge d_{p,a,q}) \rightarrow x_{ua,q} \tag{3.15}$$

$$\left(\bigwedge_{u \in S_+} \bigwedge_{q \in Q} x_{u,q} \rightarrow f_q \right) \wedge \left(\bigwedge_{u \in S_-} \bigwedge_{q \in Q} x_{u,q} \rightarrow \neg f_q \right) \tag{3.16}$$

Additionally, Heule and Verwer add the formulas

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigvee_{q \in Q} d_{p,a,q}, \tag{3.17}$$

$$\bigwedge_{u \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{q,q' \in Q, q \neq q'} \neg x_{u,q} \vee \neg x_{u,q'}, \tag{3.18}$$

$$\bigwedge_{ua \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} (x_{u,p} \wedge x_{ua,q}) \rightarrow d_{p,a,q} \tag{3.19}$$

in order to break symmetries in possible solutions and to potentially speed up the solving process. These formulas have the additional effect that the variables $d_{p,a,q}$ now encode a total function and that exactly one of the variables $x_{u,q_0}, \dots, x_{u,q_{n-1}}$ for each $u \in \mathrm{Pref}(S_+ \cup S_-)$ is set to *true*. Note that Formulas 3.15, 3.16, and 3.19 can easily be rewritten in conjunctive normal form (exploiting that $x \rightarrow y$ is equivalent to $\neg x \vee y$).

Let $\mu_n^{\mathcal{S}}(\overline{x}, \overline{d}, \overline{f})$ be the conjunction of Formulas 3.13 to 3.19 where $\overline{x}$ is the list of all variables $x_{u,q}$, $\overline{d}$ is the list of all variables $d_{p,a,q}$, and $\overline{f}$ is the list of all variables f_q with $u \in \mathrm{Pref}(S_+ \cup S_-)$, $a \in \Sigma$, and $p, q \in Q$. Given a model $\mathfrak{M}$ of $\mu_n^{\mathcal{S}}$, it is straightforward how to derive a DFA $\mathcal{A}_{\mathfrak{M}}$ (that is consistent with $\mathcal{S}$).

Definition 3.5. Let $\mathfrak{M} \models \mu_n^{\mathcal{S}}(\overline{x}, \overline{d}, \overline{f})$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0^{\mathfrak{M}}, \delta, F)$ is defined by

- $Q = \{q_0, \dots, q_{n-1}\}$;
- $q_0^{\mathfrak{M}} = q_i$ if and only if $x_{\varepsilon,q_i}^{\mathfrak{M}} = true$;
- $\delta(q_i, a) = q_j$ if and only if $d_{q_i,a,q_j}^{\mathfrak{M}} = true$ where $q_i, q_j \in Q$ and $a \in \Sigma$; and
- $F = \{q_i \in Q \mid f_{q_i}^{\mathfrak{M}} = true\}$.

The next theorem states that Heule and Verwer's method produces a minimal consistent DFA.

Theorem 3.5 (Heule and Verwer [HV10])**.** *For a sample $\mathcal{S}$ and $n \in \mathbb{N}$, a model $\mathfrak{M}$ of $\mu_n^{\mathcal{S}}(\overline{x}, \overline{d}, \overline{f})$ yields a DFA $\mathcal{A}_{\mathfrak{M}}$ with n states that is consistent with $\mathcal{S}$. A (binary) search can be used to find the minimal n for which $\mu_n^{\mathcal{S}}(\overline{x}, \overline{d}, \overline{f})$ is satisfiable, which yields a minimal DFA that is consistent with $\mathcal{S}$.*

Proof of Theorem 3.5. To proof Theorem 3.5, we follow the strategy as with the previous proofs: we show that $\mathcal{A}_{\mathfrak{M}}$ is well-defined, then prove that the variables $x_{u,q}$ encode runs of $\mathcal{A}_{\mathfrak{M}}$, and show that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$. Finally, we argue that a minimal value for n results in a minimal consistent DFA. Let $\mathfrak{M} \models \mu_n^{\mathcal{S}}(\overline{d}, \overline{f}, \overline{x})$.

Our first task is to show that $\mathcal{A}_{\mathfrak{M}}$ is well-defined. The initial state is well-defined since Formulas 3.14 and 3.18 ensure that exactly one of the variables $x_{\varepsilon,q_0}, \dots, x_{\varepsilon,q_{n-1}}$ is set to *true*. Moreover, Formulas 3.13 and 3.17 make sure that the variables $d_{p,a,q}$ encode a total deterministic transition function and, hence, δ is well-defined. Finally, the encoding of final states as Boolean variables f_q makes it obvious that F is well-defined.

To prove that the variables $x_{u,q}$ encode runs of $\mathcal{A}_{\mathfrak{M}}$ on words $u \in \mathrm{Pref}(S_+ \cup S_-)$, we show by induction over the length of u that $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u} q$ implies $x_{u,q}^{\mathfrak{M}} = true$. In other words, the variable $x_{u,q}$ signals that $\mathcal{A}_{\mathfrak{M}}$ reaches state q after reading u.

Base case Let $u = \varepsilon$. By definition of runs, we have $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{\varepsilon} q_0^{\mathfrak{M}}$. Furthermore, we know $q_0^{\mathfrak{M}} = q_i$ if and only if $x_{\varepsilon,q_i}^{\mathfrak{M}} = true$ by construction of $\mathcal{A}_{\mathfrak{M}}$, which proves the claim.

Induction step Let $u = va$ with $a \in \Sigma$, and assume $\mathcal{A}_{\mathfrak{M}}\colon q_0^{\mathfrak{M}} \xrightarrow{v} p \xrightarrow{a} q$. Applying the induction hypothesis for v yields $x_{v,p}^{\mathfrak{M}} = true$. Moreover, the transition $\delta(p,a) = q$ is applied in the last step of the run of $\mathcal{A}_{\mathfrak{M}}$ on u. Hence, $d_{p,a,q}^{\mathfrak{M}} = true$ holds by construction of $\mathcal{A}_{\mathfrak{M}}$. By considering Formula 3.15, we finally deduce that $x_{va,q}^{\mathfrak{M}} = x_{u,q}^{\mathfrak{M}} = true$ is satisfied.

We can now show that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$. To this end, let $u \in S^+$. Then, there exists a $q \in Q$ such that $\mathcal{A}_{\mathfrak{M}}\colon q_0^{\mathfrak{M}} \xrightarrow{u} q$, and, thus, $x_{u,q}^{\mathfrak{M}} = true$. In this case, Formula (3.16) ensures $f_q^{\mathfrak{M}} = true$ and, hence, $q \in F$ by definition of $\mathcal{A}_{\mathfrak{M}}$. This in turn means that $u \in L(\mathcal{A}_{\mathfrak{M}})$. Since $u \in S_+$ was chosen arbitrarily, we obtain $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$. Analogously, we deduce $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.

Finally, a model for the minimal value of n yields a minimal DFA that is consistent with $\mathcal{S}$ because one can construct a model of $\mu_n^{\mathcal{S}}$ from every consistent DFA with n states. □

3.1.5 An SMT-based Method

We recently proposed a passive learning algorithm that is based on the logic of uninterpreted functions over the natural numbers rather than on Propositional Boolean Logic (see Neider and Jansen [NJ13]). In fact, uninterpreted functions offer a natural and convenient way to encode the components of the prospective DFA.

To simplify the following description, we assume without loss of generality that the input alphabet is $\Sigma = [m]$ and the state set of the prospective DFA is $Q = [n]$ where $m, n \in \mathbb{N}_+$. Moreover, we assume that the words in $\mathrm{Pref}(S_+ \cup S_-)$ are enumerated, say $\mathrm{Pref}(S_+ \cup S_-) = \{u_0, \ldots, u_{k-1}\}$ with $u_0 = \varepsilon$.

Our method replaces the Boolean variables $d_{p,a,q}$ and f_q by two uninterpreted functions $d\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $f\colon \mathbb{N} \to \mathbb{N}$ to encode the prospective DFA. The function d encodes the transitions: the first argument of d is the source-state, the second argument the input-symbol, and the function value is the destination-state. The function f indicates whether a state is final or not; since f needs to "simulate" Boolean values for this purpose, we interpret 0 as *false* and all other values as *true*. Note that the arguments of d and f can be any natural number, but we do not use function values for arguments that are out of range; thus, a solver may choose them arbitrarily.

Additionally, we introduce an uninterpreted function $x\colon \mathbb{N} \to \mathbb{N}$ to encode the runs of the prospective DFA on the words of the sample. More precisely, $x(i) = j$ signals that the prospective DFA reaches state j after reading the word u_i. Also here, we do not care about the function value of x if the argument is not an element of $[k]$.

The following formulas encode the passive learning task in the logic of uninterpreted functions over the natural numbers.

$$x(0) < n \tag{3.20}$$

$$\bigwedge_{i \in Q} \bigwedge_{a \in \Sigma} d(i,a) < n \tag{3.21}$$

$$\bigwedge_{u_i, u_j \in \mathrm{Pref}(S_+ \cup S_-), a \in \Sigma, u_j = u_i a} x(j) = d(x(i), a) \tag{3.22}$$

$$\left(\bigwedge_{u_i \in S_+} f(x(i)) \neq 0 \right) \wedge \left(\bigwedge_{u_i \in S_-} f(x(i)) = 0 \right) \tag{3.23}$$

Formula 3.20 constrains the value representing the initial state (i.e., the state reached after reading the input $u_0 = \varepsilon$) to be an element of Q. Moreover, Formula 3.21 makes sure that the uninterpreted function d indeed represents a transition function by constraining the function values to be an element of Q. Formula 3.22 describes how the prospective DFA proceeds when processing words from $\mathrm{Pref}(S_+ \cup S_-)$. Finally, Formula 3.23 ensures that the prospective DFA reaches a final state after reading a word from S_+ and a nonfinal state after reading a word from S_-.

Let $\nu_n^{\mathcal{S}}(x,d,f)$ be the conjunction of Formulas 3.20 to 3.23 where x, d, and f are the uninterpreted functions described above. Given a model $\mathfrak{M} \models \nu_n^{\mathcal{S}}(x,d,f)$, one can easily construct a DFA $\mathcal{A}_{\mathfrak{M}}$ (that is consistent with $\mathcal{S}$) as described next.

Definition 3.6. Let $\mathfrak{M} \models \nu_n^{\mathcal{S}}(x,d,f)$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0^{\mathfrak{M}}, \delta, F)$ is defined by

- $Q = [n]$;
- $q_0^{\mathfrak{M}} = x^{\mathfrak{M}}(0)$;
- $\delta(i,a) = d^{\mathfrak{M}}(i,a)$ where $i \in Q$ and $a \in \Sigma$; and
- $F = \{i \in Q \mid f^{\mathfrak{M}}(i) \neq 0\}$.

The following theorem states that the SMT-based method indeed produces a minimal consistent DFA.

Theorem 3.6. *For a sample $\mathcal{S}$ and $n \in \mathbb{N}$, a model $\mathfrak{M}$ of $\nu_n^{\mathcal{S}}(x,d,f)$ yields a DFA $\mathcal{A}_{\mathfrak{M}}$ with n states that is consistent with $\mathcal{S}$. A (binary) search can be used to find the smallest n for which $\nu_n^{\mathcal{S}}(x,d,f)$ is satisfiable, which yields a minimal DFA that is consistent with $\mathcal{S}$.*

Proof of Theorem 3.6. We follow our proof strategy once more: we show that $\mathcal{A}_{\mathfrak{M}}$ is well-defined, then prove that the function $x^{\mathfrak{M}}$ encodes runs of $\mathcal{A}_{\mathfrak{M}}$, and show that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$. Finally, we argue that a minimal value for n results in a minimal consistent DFA. Let $\mathfrak{M} \models \nu_n^{\mathcal{S}}(x,d,f)$.

We first note that the initial state of $\mathcal{A}_{\mathfrak{M}}$ is well-defined since Formula 3.20 ensures that the value of $x^{\mathfrak{M}}(0)$ ranges in Q and, hence, is a valid state. Moreover, Formula 3.21 ensures that $d^{\mathfrak{M}}(q,a)$ ranges in Q for all $p \in Q$ and $a \in \Sigma$. Thus, δ is well-defined. Finally, it is easy to see that F is well-defined, too.

To prove that the function $x^{\mathfrak{M}}$ encodes valid runs, we show by induction over the length of words $u_\ell \in \mathrm{Pref}(S_+ \cup S_-)$ that $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u_\ell} j$ implies $x^{\mathfrak{M}}(\ell) = j$.

Base case Let $u = u_0 = \varepsilon$. By definition of runs, we have $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{\varepsilon} q_0^{\mathfrak{M}}$. Moreover, $q_0^{\mathfrak{M}} = x^{\mathfrak{M}}(0)$ holds by definition of $\mathcal{A}_{\mathfrak{M}}$.

Induction step Let $u_\ell = u_k a$ and consider the run $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u_k} i \xrightarrow{a} j$. Applying the induction hypothesis yields $i = x^{\mathfrak{M}}(k)$. Moreover, the transition $\delta(i,a) = j$ is applied in the last step of the run. Thus, $d^{\mathfrak{M}}(i,a) = j$ is satisfied due to the construction of $\mathcal{A}_{\mathfrak{M}}$. Formula 3.22 now enforces that then $x^{\mathfrak{M}}(\ell) = j$ holds.

Next, we show that $\mathcal{A}_{\mathfrak{M}}$ is consistent with $\mathcal{S}$. To this end, let $u_\ell \in S^+$. Then, there exists an $i \in Q$ such that $\mathcal{A}_{\mathfrak{M}} \colon q_0^{\mathfrak{M}} \xrightarrow{u_\ell} i$, and, thus, $x^{\mathfrak{M}}(\ell) = i$. In this case, Formula 3.23 ensures $f^{\mathfrak{M}}(i) \neq 0$, which implies $i \in F$ by definition of $\mathcal{A}_{\mathfrak{M}}$. This in turn means that $u_\ell \in L(\mathcal{A}_{\mathfrak{M}})$. Since $u_\ell \in S_+$ was chosen arbitrarily, we obtain $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$. Analogously, we deduce $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.

We conclude the proof by stating that a model for the minimal n yields a minimal DFA that is consistent with $\mathcal{S}$ because one can construct a model of $\varphi_n^{\mathcal{S}}$ from every consistent DFA with n states. □

3.1.6 Evaluation

We conclude the section on passive learning with both a theoretical comparison (with respect to the size of the resulting formulas, the number of variables, and so on) and an experimental benchmark (which serves to demonstrate how a particular method performs in practice). The motivation for the latter is that a comparison merely based on properties of the generated logic formulas is misleading. On the one hand, a larger formula potentially contains additional information that might help the solver to find a solution faster; a prototypical and well-studied example is adding clauses to formulas in Propositional Boolean Logic to break symmetries (e.g., discussed by Aloul et al. [ARMS03]). On the other hand, the actual performance of different logic solvers can vary considerably.

It is worth noting that Grinchtein, Leucker, and Piterman [GLP06] as well as Heule and Verwer [HV10] conducted similar experimental comparisons. These comparisons, however, cover only a part of the methods presented in this chapter.

Table 3.2: Overview of the passive learning algorithms described in Section 3.1 with respect to the used logics and the generated formulas. The parameter n refers to the size of the prospective DFA, and $k = |\mathrm{Pref}(S_+ \cup S_-)|$.

Method	Logic	Size of formula
Biermann and Feldman	Equality Logic	$\mathcal{O}(k)$ variables, $\mathcal{O}(k + \lvert S_+\rvert \cdot \lvert S_-\rvert)$ constraints
Grinchtein, Leucker, and Piterman (unary)	Propositional Boolean Logic	$\mathcal{O}(kn)$ variables, $\mathcal{O}(kn^2 + n\lvert S_+\rvert \cdot \lvert S_-\rvert)$ clauses
Grinchtein, Leucker, and Piterman (binary)	Propositional Boolean Logic	$\mathcal{O}(k \log_2 n)$ variables, $\mathcal{O}(kn \log_2 n + n\lvert S_+\rvert \cdot \lvert S_-\rvert)$ clauses
Heule and Verwer	Propositional Boolean Logic	$\mathcal{O}(n^2\lvert\Sigma\rvert + kn)$ variables, $\mathcal{O}(n^3\lvert\Sigma\rvert + kn^2)$ clauses
SMT-based	Logic of uninterpreted functions over $\mathbb{N}$	3 uninterpreted functions, $\mathcal{O}(n\lvert\Sigma\rvert + k)$ constraints

Theoretical Comparison

Table 3.2 summarizes the key properties of the passive learning algorithms presented in this section. For each method, the table lists the underlying logic and the asymptotic size of the logic formulas that need to be solved during the learning process. The parameter n refers to the size of the prospective DFA, and $k = |\mathrm{Pref}(S_+ \cup S_-)|$ is a measure for the amount of information contained in the given sample. Recall that all methods described here use a search with parameter n of some kind to determine a minimal consistent DFA.

Comparing the size of formulas in different logics is clearly problematic, but we can make two observations regarding the methods based on Propositional Boolean Logic: first, Grinchtein, Leucker, and Piterman's method based on the binary encoding results in smaller formulas than their method based on the unary encoding; second, the formulas generated by Heule and Verwer's method depend on the alphabet, whereas the formulas generated by Grinchtein, Leucker, and Piterman's methods do not. The latter observation might be of special interest as it suggests that Grinchtein, Leucker, and Piterman's methods should work better in situation in which the alphabet is very large compared to the size of the sample.

Experimental Comparison

For the experimental comparison, we implemented the presented methods and benchmarked them as described next.

Methodology We evaluated the passive learning methods on two different benchmark suites.

The first benchmark suite was already used by Heule and Verwer [HV10].[3] The benchmark suite consists of 810 samples, each of which contains between 517 and 550 positively and negatively classified words over the alphabet $\{0,1\}$. A feature of this benchmark suite is that all samples are prefix-closed.

The second benchmark suite consists of *random samples*, which we generated according to the following procedure:

1. We fixed four parameters $k, m, n \in \mathbb{N}_+$ and $p \in \mathbb{R}$ with $0 < p < 1$.

2. We generated a random DFA $\mathcal{A}$ with n states over the alphabet $\Sigma = [m]$ according to method developed by Champarnaud and Paranthoën [CP05]. Broadly speaking, their method produces randomly drawn DFAs of size n over the alphabet $\Sigma = [m]$ such that "most" of these DFAs are minimal with respect to the language they accept. Champarnaud and Paranthoën's method is implemented in Libalf.

3. We generated k distinct words $u_1, \dots, u_k$ as follows: first, we determined the length of each word using a geometric distribution with parameter p; second, we constructed $u_i = a_1 \dots a_{\ell_i}$ by choosing $a_j \in [m]$ uniformly randomly for each $j \in \{1, \dots, \ell_i\}$.

4. We classified the words $u_1, \dots, u_k$ with respect to $\mathcal{A}$; that is, we generated the sample $\mathcal{S} = (S_+, S_-)$ such that $S_+ = \{u_1, \dots, u_k\} \cap L(\mathcal{A})$ and $S_- = \{u_1, \dots, u_k\} \setminus L(\mathcal{A})$. By using $\mathcal{A}$ to classify the samples, we made sure that a consistent DFA with at most n states is guaranteed to exist.

We fixed $k = 150$, $n = 10$, and $p = 0.2$ and generated three sets of samples according to the procedure described above, each of which consisting of 250 samples; we fixed $m = 2$ for the first set, $m = 5$ for the second set, and $m = 10$ for the third set. Thus, our second benchmark suite contains 750 samples in total.

We implemented the presented passive learning algorithms in Libalf. We used the Glucoser SAT solver for Grinchtein, Leucker, and Piterman's as well as Heule and Verwer's methods, and Microsoft's Z3 SMT solver for Biermann and Feldman's as well as the SMT-based method. In all cases, we implemented the following search scheme: we start with $n = 1$ and successively increase n by one until a minimal consistent DFA is found. Since fixing variables for obviously different words gives the same additional knowledge to every method, we decided to not implement this feature in order to keep the implementation as simple as possible.

[3] As to our knowledge, this benchmark suite originates from Oliveira and Silva [OS98].

All experiments were performed on an Intel Q9550 quad core CPU at 2.83 GHz with 4 GiB of RAM running Ubuntu 12.04 LTS. Our implementation is not multi-threaded and used a single core.

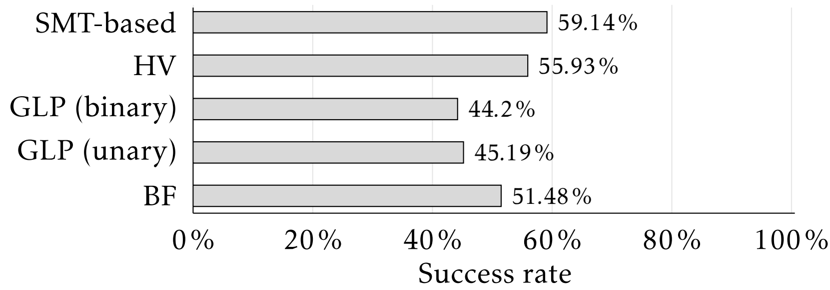
Since all passive learning algorithms produce minimal consistent DFAs, runtime and memory consumption are the only interesting measures. To make benchmarking a large number of samples possible and to guarantee equal conditions for all methods, we imposed a runtime limit of 300 s and a memory limit of 3.5 GiB.

Results Figure 3.2 presents the results of our experiments. "HV" refers to Heule and Verwer's method, "GLP" to Grinchtein, Leucker, and Piterman's methods, and "BF" to Biermann and Feldman's method.; "Success rate" is the quotient of the number of samples for which the respective method found a minimal consistent DFA within 300 seconds and the total number of samples in the benchmark suite.

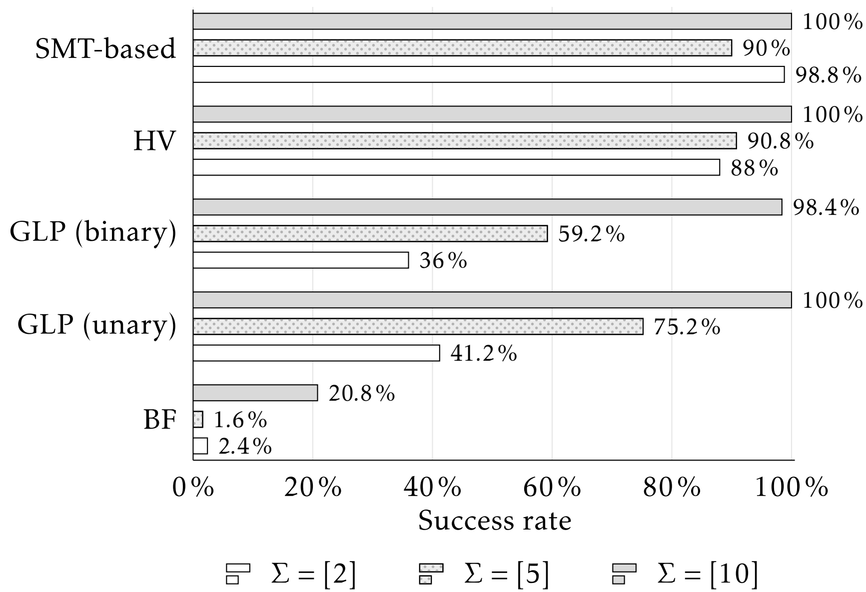
Figure 3.2a shows the results on Heule and Verwer's benchmark suite. The success rate of all methods lay within a narrow band. The SMT-based method performed best, closely followed by Heule and Verwer's as well as Biermann and Feldman's method. Both of Grinchtein, Leucker, and Piterman's methods performed almost equally and ranked last.

Figure 3.2b depicts the results on the second benchmark suite. All methods except for Biermann and Feldman's succeeded in learning a minimal consistent DFA for almost all random samples over the alphabet $\Sigma = [10]$; Biermann and Feldman's method only succeeded in about one out of five of these samples. However, the results on random samples over the alphabets $\Sigma = [2]$ and $\Sigma = [5]$ are more diverse: the SMT-based method and Heule and Verwer's method performed best; Grinchtein, Leucker and Piterman's unary encoding ranked third, followed by Grinchtein, Leucker and Piterman's binary encoding; Biermann and Feldman's method failed on almost all samples and ranked last.

Discussion The main result of our experimental evaluation is that the SMT-based method and Heule and Verwer's method excel the other methods in all of our experiments (except for the experiments of the second benchmark suite with $\Sigma = [10]$). Biermann and Feldman's method proved to be competitive on the first benchmark suite but performed poorly on all random samples. Also, both of Grinchtein, Leucker and Piterman's methods were unable to find solutions within the runtime limit for more than half of all random samples over the alphabet $\Sigma = [2]$. In total, our experiments suggest that the SMT-based method and Heule and Verwer's method should be preferred in applications. However, we want to point out that our findings differ from both Grinchtein, Leucker, and Piterman's as well as Heule and Verwer's.



(a) Results on Heule and Verwer's benchmark suite.



(b) Results on random samples.

Figure 3.2: Performance of the passive learning algorithms described in Section 3.1 on both benchmark suites. "HV" refers to Heule and Verwer's method; "GLP" refers to Grinchtein, Leucker, and Piterman's methods; "BF" refers to Biermann and Feldman's method.

Grinchtein, Leucker, and Piterman [GLP06] report superior performance of the unary encoding compared to the binary encoding, although in a different setting where the passive learning is embedded in an active learning scenario. Their explanation for this behavior is that in the case of the unary encoding, "the information encoded in the SAT problem is less 'packed' allowing a SAT solver to perform more optimizations" [GLP06, Page 496]. However, our experiments did not show a substantial difference between the unary and the binary encoding, neither on the first nor on the second benchmark suite.

Heule and Verwer [HV10] evaluated their method as well as Grinchtein, Leucker, and Piterman's on the samples of the first benchmark suite. Heule and Verwer report that their method found minimal consistent DFAs for all 810 samples in less than 200 seconds. This disagrees with our findings: in our experiments, Heule and Verwer's method solved about half of these samples. Moreover, the authors report a "huge difference" [HV10, Page 76] between their method and Grinchtein, Leucker, and Piterman's, which we could also not observe. However, it is likely that these discrepancies have to be attributed to the use of different SAT solvers (Glucoser used in our experiments, zChaff [MMZ+01] used by Grinchtein, Leucker, and Piterman, and PicoSat [Bie08] used by Heule and Verwer).

Finally, let us comment on the fact that finding minimal consistent DFAs for larger alphabets turned out to be simpler than for small alphabets. A detailed analysis of the experimental results shows that samples over large alphabets often allow for "small" consistent DFAs and, thus, many methods are able to find a solution. A likely explanation for this is that only "few" words in samples over large alphabets share a common suffix and, thus, need to be distinguished by different states. By contrast, a sample of the same size over a small alphabet is likely to contain "many" words with common suffixes, which requires more states to distinguish such words.

3.2 Active Learning of DFAs

The probably most famous learning setting is the active learning setting introduced by Angluin [Ang87]. In this setting, a learning algorithm—often called *learner*—learns a regular *target language* $L \subseteq \Sigma^*$ over an a priori fixed alphabet Σ by actively querying a *teacher*. The teacher has access to the language in question and can answer two different types of queries: membership and equivalence queries.

Membership query On a membership query, the learner provides a word $u \in \Sigma^*$, and the teacher replies "yes" or "no" depending on whether $u \in L$ or not.

Equivalence query On an equivalence query, the learner conjectures a regular language, typically given as a DFA $\mathcal{A}$, and the teacher checks whether $\mathcal{A}$ is an equivalent

description of the target language. Is this the case, then the teacher replies "yes"; otherwise, the teacher returns a so-called *counterexample* $u \in \Sigma^*$ with $u \in L(\mathcal{A}) \Leftrightarrow u \notin L$ as a witness that $L(\mathcal{A})$ and L are different.

The task associated with this learning setup is the following.

Definition 3.7 (Active learning of DFAs)**.** Given a teacher capable of answering membership and equivalence queries, the *active learning task* is to compute a DFA of minimal size that passes an equivalence query.

Note that active learning in the sense of Definition 3.7 guarantees that the resulting DFA is unique since there exists a unique minimal DFA for each regular language.

A naïve algorithm can already solve the active learning task: since DFAs over a fixed alphabet can be enumerated according to their size, a learning algorithm can successively ask equivalence queries with DFAs from this enumeration until a DFA passes the query. This shows that active learning can be done in a much simpler setup that only allows equivalence queries without counterexamples. However, such a naïve algorithm is surely infeasible in practice, and a learning algorithm should use counterexamples and membership queries to improve performance.

Several learning algorithms for the learning task of Definition 3.7 have been developed. These algorithms typically run in time polynomial in two parameters: the size of the minimal DFA recognizing the target language, which measures the complexity of the target language, and the length of the longest counterexample returned by the teacher, which measures the (in)efficiency of the teacher. In the remainder of this section, we describe two popular algorithms in detail: Angluin's algorithm [Ang87] in Section 3.2.1 and Kearns and Vazirani's algorithm [KV94] in Section 3.2.2. We conclude in Section 3.2.3 with a cursory description of a third learning algorithm due to Rivest and Schapire [RS93] and a comparison of the presented algorithms.

For the remaining section, fix an alphabet Σ and a regular target language $L \subseteq \Sigma^*$. In addition, let $\mathcal{A}_L = (Q_L, \Sigma, q_0^L, \delta_L, F_L)$ be the canonical minimal DFA accepting L as defined in Section 2.1.

3.2.1 Angluin's Learning Algorithm

Angluin [Ang87] has not only introduced the active learning setting but has also provided an appropriate learning algorithm. Angluin's algorithm works by approximating the Nerode congruence $\sim_L$ of the target language L: starting with a coarse approximation, the algorithm refines this approximation successively until the Nerode congruence is computed exactly. This procedure then yields a DFA that is isomorphic to the canonical minimal DFA of the target language.

Angluin's algorithm stores the information gathered during the learning process in a so-called *observation table* $O = (R, S, T)$ where $R \subseteq \Sigma^*$ is a prefix-closed set of *representatives*, $S \subseteq \Sigma^*$ is a set of *separating words*, and $T\colon (R \cup R \cdot \Sigma) \cdot S \to \{0, 1\}$ is a mapping that stores the table entries. Intuitively, R contains candidates for representatives of the L-equivalence classes, whereas S contains words to distinguish the representatives. Angluin's algorithm makes sure that the data stored in the table always agrees with the target language (i.e., it maintains $T(u) = 1$ if the teacher replies "yes" to a membership query on u and $T(u) = 0$ if the teacher replies "no"); in other words, the algorithm maintains $T(u) = 1$ if and only if $u \in L$ for all $u \in (R \cup R \cdot \Sigma) \cdot S$.

Given an observation table O, two words $u, v \in R \cup R \cdot \Sigma$ are *O-equivalent*, denoted by $u \sim_O v$, if $T(uw) = T(vw)$ holds for all separating words $w \in S$. For brevity, we denote the O-equivalence class of a word $u \in R \cup R \cdot \Sigma$ by $[\![u]\!]_O = \{v \in R \cup R \cdot \Sigma \mid u \sim_O v\}$ (rather than $[\![u]\!]_{\sim_O}$).

It is not hard to verify that two L-equivalent words $u, v \in R \cup R \cdot \Sigma$ are always O-equivalent because $\sim_O$ separates two words by only using the separating words in S. More precisely, if $u \sim_L v$, then $uw \in L$ holds if and only if $vw \in L$ holds for all $w \in \Sigma^*$. Since this is in particular true for all $w \in S$, $T(uw) = T(vw)$ holds for all $w \in S$. Thus, $u \sim_L v$ implies $u \sim_O v$. This in turn implies $\text{index}(\sim_O) \leq \text{index}(\sim_L)$.

The key idea of Angluin's algorithm is to make the observation table *closed* and *consistent*.

- An observation table O is *closed* if for all $u \in R$ and $a \in \Sigma$ some $v \in R$ with $ua \sim_O v$ exists. If O is not closed, then Angluin's algorithm adds ua to R and updates O using membership queries. Note that R stays prefix-closed after adding ua because R was prefix-closed before and u belongs to R. In addition, O does no longer violate the closedness property for u and a (although the table might still not be closed).

- An observation table is *consistent* if $u \sim_O v$ implies $ua \sim_O va$ for all $u, v \in R$ and $a \in \Sigma$. If O is not consistent, then there exist $u, v \in R$, $a \in \Sigma$, and $w \in S$ with $u \sim_O v$ and $T(uaw) \neq T(vaw)$. In this case, Angluin's algorithm adds aw to S and updates O using membership queries. Now, the observation table no longer violates the consistency property for u and v with respect to a (although the table might still not be consistent).

Once the observation table is both closed and consistent, Angluin's algorithm derives a conjecture $\mathcal{A}_O$ from the observation table. Similar to the construction of a DFA from the Nerode congruence of a regular language, the O-equivalence classes of representatives in R form the states of $\mathcal{A}_O$ and $\sim_O$ determines the transitions. More precisely, Angluin's algorithm constructs the DFA $\mathcal{A}_O = (Q_O, \Sigma, q_0^O, \delta_O, F_O)$ where

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

It is not hard to verify that $\mathcal{A}_O$ is well-defined: The initial state is well-defined since R is prefix-closed and, hence, $\varepsilon \in R$. Moreover, δ_O is well defined since O is closed and consistent (i.e., $\delta([\![u]\!]_O, a)$ is well-defined for all $u \in R$ and independent of the representative $v \in [\![u]\!]_O$). The following lemma states that this construction results in a DFA that works correctly on all representatives.

Lemma 3.7 (Angluin [Ang87]). *Let $O = (R, S, T)$ be a closed and consistent observation table for a regular language $L \subseteq \Sigma^*$ and $\mathcal{A}_O$ as defined above. Then, $\mathcal{A}_O$ is correct on all representatives $u \in R$ (i.e., $\mathcal{A}_O$ satisfies $u \in L(\mathcal{A}_O)$ if and only if $u \in L$ for all $u \in R$).*

Proof of Lemma 3.7. We first show that $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{u} [\![u]\!]_O$ holds for all $u \in R$ using an induction over the length of the representative u. Note that all prefixes of u also belong to R since R is prefix-closed.

Base case Let $u = \varepsilon$. Then, $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{\varepsilon} [\![\varepsilon]\!]_O$ holds by definition of runs.

Induction step Let $u = va$. By induction hypothesis, we obtain $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{v} [\![v]\!]_O$. Moreover, $\delta([\![v]\!]_O, a) = [\![va]\!]_O = [\![u]\!]_O$ holds by definition of δ_O since $va = u \in R$. Thus, $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{v} [\![v]\!]_O \xrightarrow{a} [\![va]\!]_O$ with $[\![va]\!]_O = [\![u]\!]_O$.

Now, consider a representative $u \in R$. Since $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{u} [\![u]\!]_O$ and by definition of F_O, we obtain

$$\begin{aligned} u \in L(\mathcal{A}_O) &\Leftrightarrow \mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{u} [\![u]\!]_O \text{ and } [\![u]\!]_O \in F_O \\ &\Leftrightarrow T(u) = 1 \\ &\Leftrightarrow u \in L. \end{aligned}$$

□

In addition to Lemma 3.7, $\mathcal{A}_O$ is isomorphic to $\mathcal{A}_L$ if O contains enough information in that at least one representative of every $\sim_L$-equivalence class is present in R (i.e, $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$ is satisfied). Note that the restriction $\sim_O \cap R \times R$ of $\sim_O$ on the set R of representatives still is an equivalence relation.

Lemma 3.8 (Angluin [Ang87]). *Let $O = (R, S, T)$ be a closed and consistent observation table for a regular language $L \subseteq \Sigma^*$. If $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$, then $\mathcal{A}_O$ is isomorphic to $\mathcal{A}_L$.*

Proof of Lemma 3.8. Let O be closed and consistent, $\mathrm{index}(\sim_O \cap R \times R) = \mathrm{index}(\sim_L)$, and $\mathcal{A}_O$ be the DFA derived from O.

We first show by induction over the length of words $u \in \Sigma^*$ that the run of $\mathcal{A}_O$ on u is $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{u} [\![u']\!]_O$ with $u' \in R$ and $u' \sim_L u$. This fact implies $L(\mathcal{A}_O) = L$ because

$$\begin{aligned} u \in L(\mathcal{A}_O) &\Leftrightarrow \mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{u} [\![u']\!]_O \text{ with } u' \in R \text{ and } [\![u']\!]_O \in F_O \\ &\Leftrightarrow T(u') = 1 \\ &\Leftrightarrow u' \in L \\ &\Leftrightarrow u \in L. \end{aligned}$$

Base case Let $u = \varepsilon$. Then, $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{\varepsilon} [\![\varepsilon]\!]_O$ holds by definition of runs. Since $\sim_L$ is a congruence (and in particular reflexive), $\varepsilon \sim_L \varepsilon$ holds.

Induction step Let $u = va$ and $\mathcal{A}_O \colon [\![\varepsilon]\!]_O \xrightarrow{v} [\![v']\!]_O \xrightarrow{a} [\![v'']\!]_O$ be the run of $\mathcal{A}_O$ on u where $v', v'' \in R$. Applying the induction hypothesis now yields $v' \sim_L v$. Since $\sim_L$ is a congruence, we also know that $v'a \sim_L va$ holds. Furthermore, $v'a \sim_O v''$ holds by construction of $\mathcal{A}_O$ because the transition $\delta_O([\![v']\!]_O, a) = [\![v'']\!]_O$ was used in the run. Since $\mathrm{index}(\sim_O \cap R \times R) = \mathrm{index}(\sim_L)$ (and since $u \sim_L v$ implies $u \sim_O v$ for all $u, v \in R$), we obtain $v'a \sim_L v''$. In total, this shows $u = va \sim_L v'a \sim_L v''$.

In conclusion, the fact that $\mathcal{A}_O$ is isomorphic to $\mathcal{A}_L$ follows from $L(\mathcal{A}_O) = L$ and $|\mathcal{A}_O| = \mathrm{index}(\sim_O \cap R \times R) = \mathrm{index}(\sim_L) = |\mathcal{A}_L|$. □

Algorithm 3.1 presents Angluin's algorithm in pseudo code. The algorithm proceeds in rounds: in every round, Angluin's algorithm makes the table closed and consistent; once this is the case, Angluin's algorithm constructs a conjecture, which it submits to an equivalence query. The learning terminates once the teacher replies "yes" on an equivalence query. However, if the teacher returns a counterexample $u \in \Sigma^*$, the algorithm adds u and all of its prefixes to R and updates the table. By adding all prefixes of u as new representatives, Angluin's algorithm makes sure that there is enough additional information to discover a new L-equivalence class. Once the table has been updated, the algorithm continues with the next iteration.

After every extension of the table, the function *update*(O) conducts a membership query for every table entry $u \in (R \cup R \cdot \Sigma) \cdot S$ for which no membership information is yet present: if the teacher replies "yes" on a membership query with u, it sets $T(u) = 1$; if the answer is "no", it sets $T(u) = 0$. This way, the table entries always agree with the target language.

To gain a better understanding of Angluin's algorithm, the following example illustrates its functioning.

Algorithm 3.1: Angluin's active learning algorithm.

Input: A teacher for a regular language $L \subseteq \Sigma^*$.

```
1  Initialize the observation table O = (R, S, T) with R = S = {ε} and update(O).
2  repeat
3      while O is not closed or not consistent do
4          if O is not closed then
5              Pick u ∈ R and a ∈ Σ with [[ua]]_O ∩ R = ∅.
6              R ← R ∪ {ua}.
7              update(O).
8          end
9          if O is not consistent then
10             Pick u ~_O v ∈ R, a ∈ Σ, and w ∈ S with T(uaw) ≠ T(vaw).
11             S ← S ∪ {aw}.
12             update(O).
13         end
14     end
15     Construct A_O and perform an equivalence query with A_O.
16     if the teacher returns a counterexample u then
17         R ← R ∪ Pref(u).
18         update(O).
19     end
20 until the teacher replies "yes" to the equivalence query with A_O.
21 return A_O.
```

O_0	ε
ε	0
a	0
b	1

O_1	ε
ε	0
b	1
a	0
ba	0
bb	0

O_2	ε
ε	0
b	1
bb	0
bbb	0
a	0
ba	0
bba	0
bbba	0
bbbb	0

O_3	ε	b
ε	0	1
b	1	0
bb	0	0
bbb	0	0
a	0	1
ba	0	1
bba	0	0
bbba	0	0
bbbb	0	0

Figure 3.3: The sequence of observation tables produced during the run of Angluin's algorithm in Example 3.2.

Example 3.2. Let us consider the target language $L \subseteq \{a, b\}^*$ accepted by the DFA $\mathcal{A}$ depicted below. It is not hard to verify that $\mathcal{A}$ is the minimal DFA recognizing L.

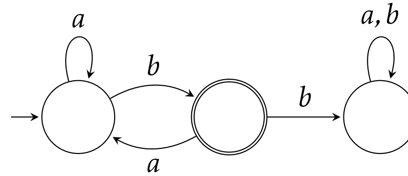
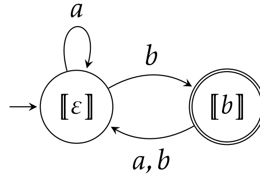


Figure 3.3 shows the sequence of observation tables that Angluin's algorithm produces in this example. The observation tables are indexed with the iteration of the outer loop of Algorithm 3.1 (Lines 2 to 20) in which they are produced. Each table is separated into an upper part and a lower part: the upper parts contain information about representatives, whereas the lower parts contain information about words from $R \cdot \Sigma$ that are no representatives themselves. The representatives are located in the first column, and the separating words are located in the first row. Both representatives and distinguishing words are printed in italics. Note that a table entry $T(u)$ might occur more than once in this depiction; for instance, the entry $T(bb)$ in the table O_3 can be found in the row of b and bb. This lengthy depiction, however, allows recognizing O-equivalent representatives easily because such representatives have identical rows.

Angluin's algorithm starts by initializing the empty observation table O_0 with $R = S = \{\varepsilon\}$. The table O_0 is not closed because there exists no representative that is O_0-equivalent to b. Hence, Angluin's algorithm adds b to R and updates the table, resulting in the observation table O_1.

The observation table O_1 is both closed and consistent, and Angluin's algorithm derives the conjecture $\mathcal{A}_{O_1}$, which is depicted below (for the sake of readability, we omit the subscripts of equivalence classes in the picture).



An equivalence query with $\mathcal{A}_{O_1}$ reveals that $\mathcal{A}_{O_1}$ is not equivalent to L, and the teacher returns a counterexample, say bbb. The algorithm now adds bbb and all of its prefixes to R and updates the table. This results in the observation table O_2.

The observation table O_2 is not consistent since $\varepsilon \sim_{O_2} bb$ but $T(\varepsilon \cdot b \cdot \varepsilon) = T(b)$ and $T(bb \cdot b \cdot \varepsilon) = T(bbb)$ disagree. Thus, Angluin's algorithm adds b to S and updates the table, resulting in the observation table O_3.

The observation table O_3 is both closed and consistent. Angluin's algorithm now derives the conjecture $\mathcal{A}_{O_3}$, which is equivalent to $\mathcal{A}$. Hence, the teacher returns "yes" on an equivalence query with $\mathcal{A}_{O_3}$. Then, Angluin's algorithm terminates and returns the DFA $\mathcal{A}_{O_3}$, which is isomorphic to $\mathcal{A}_L$. ◁

The following theorem summarizes the main properties of Angluin's algorithm.

Theorem 3.9 ([Ang87]). *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Algorithm 3.1 learns a DFA isomorphic to the canonical minimal DFA $\mathcal{A}_L$ in time polynomial in the size n of $\mathcal{A}_L$ and the length m of the longest counterexample returned by the teacher. It asks $\mathcal{O}(n)$ equivalence queries and $\mathcal{O}(mn^2)$ membership queries.*

Proof of Theorem 3.9. We split the proof into two parts. We first show that Algorithm 3.1 terminates and returns a DFA isomorphic to $\mathcal{A}_L$. Then, we estimate the number of membership and equivalence queries.

Let $n = \text{index}(\sim_L)$.

Correctness We begin by noting the following facts about the observation table $O = (R, S, T)$ during the run of Algorithm 3.1.

1. If O is not closed, Angluin's algorithm adds a new representative to R whose O-equivalence class is not yet represented by a word in R. If O is not consistent, the algorithm adds a new separating word to S, which now separates two representatives that were O-equivalent before. In both cases, $\text{index}(\sim_O \cap R \times R)$ increases.

2. After inserting a counterexample and all of its prefixes, the observation table is no longer closed, or no longer consistent, or $\text{index}(\sim_O \cap R \times R)$ increased. To prove that these are the only cases possible, we show that $\text{index}(\sim_O \cap R \times R)$

has to increase if O stays closed and consistent after processing a counterexample. Thus, let $O = (R, S, T)$ be a closed and consistent observation table, u the counterexample returned by the teacher on an equivalence query with $\mathcal{A}_O$, and $O' = (R', S', T')$ the observation table after the counterexample u has been added. Towards a contradiction, suppose that O' is closed and consistent, and assume further that $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_{O'} \cap R' \times R')$ holds. In this situation, Algorithm 3.1 produces the same conjecture again because the construction of conjectures is independent of the representatives. That means that $\mathcal{A}_O$ and $\mathcal{A}_{O'}$ are isomorphic. By Lemma 3.7, we know that $\mathcal{A}_{O'}$ works correctly on all words from R', and in particular on u. Hence, also $\mathcal{A}_O$ works correctly on u. This is a contradiction because u is a counterexample returned on an equivalence query with $\mathcal{A}_O$.

Observation 1 implies that the observation table is both closed and consistent once $\text{index}(\sim_O \cap R \times R) = n$; otherwise, Angluin's algorithm extends the table and $\text{index}(\sim_O \cap R \times R)$ increases, which contradicts $\text{index}(\sim_O \cap R \times R) \leq \text{index}(\sim_O) \leq n$. As long as $|\mathcal{A}_O| = \text{index}(\sim_O \cap R \times R) < n$, the DFA $\mathcal{A}_O$ necessarily accepts a language different from L and the teacher returns a counterexample. Moreover, Observation 2 in combination with Observation 1 implies that $\text{index}(\sim_O \cap R \times R)$ increases between consecutive equivalence queries. Thus, $\text{index}(\sim_O \cap R \times R) = n$ holds eventually. Once this is the case, the observation table is closed and consistent, and Lemma 3.8 ensures that $\mathcal{A}_O$ is isomorphic to $\mathcal{A}_L$. Then, $\mathcal{A}_O$ passes an equivalence query and Algorithm 3.1 terminates and returns a DFA that is isomorphic to $\mathcal{A}_L$.

Complexity Due to Observation 1, the observation table can be not closed or not consistent at most n times. If the table is not closed, adding a new representative to R and updating the table requires at most $|\Sigma| \cdot |S|$ membership queries. If the table is not consistent, adding a new separating word to S and updating the table requires at most $|\Sigma| \cdot |R|$ membership queries.

In addition, Algorithm 3.1 constructs at most n incorrect conjectures because $\text{index}(\sim_O \cap R \times R)$ increases between consecutive equivalence queries and is bounded by n. By adding a counterexample and its prefixes to the table, R may increase by at most m where m is the length of the longest counterexample returned by the teacher. Thus, updating the table in the case of a counterexample requires at most $m \cdot |\Sigma| \cdot |S|$ membership queries.

In summary, the size of the table is in $\mathcal{O}(mn^2)$ since $|R| \leq mn$, $|S| \leq n$, and Σ is fixed a priori. Thus, Algorithm 3.1 asks $\mathcal{O}(n)$ equivalence queries and $\mathcal{O}(mn^2)$ membership queries. Checking for closedness and consistency, extending and updating the table, as well as constructing conjectures can be done in time polynomial in the size of the

table. Hence, Algorithm 3.1 terminates in time polynomial in m and n if implemented properly. □

3.2.2 Kearns and Vazirani's Learning Algorithm

Kearns and Vazirani's learning algorithm [KV94] is another algorithm designed to learn a regular target language $L \subseteq \Sigma^*$ in Angluin's active learning setting. Like Angluin's algorithm, Kearns and Vazirani's algorithm computes an approximation of the target language's Nerode congruence, which it refines until the Nerode congruence is computed exactly. However, Kearns and Vazirani's algorithm tries to avoid unnecessary membership queries and organizes its data in a binary tree rather than a table. This tree contains exactly one representative for each L-equivalence class discovered so far and only uses a subset of separating words to witness that two representatives are not L-equivalent. Angluin's algorithm, on the other hand, potentially stores several representatives for an L-equivalence class. Moreover, it queries information for all combinations of representatives and separating words.

Recall that we assume $L \subseteq \Sigma^*$ to be a regular language, $\sim_L$ the Nerode congruence of L, and $\mathcal{A}_L$ the canonical minimal DFA accepting L. We simplify the following description somewhat by assuming $L \neq \emptyset$ and $L \neq \Sigma^*$. Kearns and Vazirani's algorithm takes care of these situations in a separate preprocessing step.

Like Angluin's algorithm, Kearns and Vazirani's learning algorithm stores its learned data in two nonempty sets $R, S \subseteq \Sigma^*$. The set R consists of *representatives* that are used to represent the equivalence classes of $\sim_L$. The algorithm ensures that all representatives are distinct in the sense that there are no two representatives in R that represent the same L-equivalence class, hence, preserving $|R| \leq \text{index}(\sim_L)$. The set S consists of *separating words* that are used to witness that two different representatives indeed represent different equivalence classes. More formally, Kearns and Vazirani's algorithm keeps a separating word $v \in S$ for any two representatives $u \neq u' \in R$ such that $uv \in L \Leftrightarrow u'v \notin L$ is satisfied. Furthermore, the algorithm guarantees $\varepsilon \in R$ (i.e., a representative for the equivalence class $[\![\varepsilon]\!]_L$ is always present) and $\varepsilon \in S$ (i.e., representatives belonging to L can be distinguished from those not belonging to L).

The algorithm organizes R and S in an ordered binary tree $t_{R,S}$ called *classification tree* (we drop the subscript if R and S are clear from the context). The inner nodes are labeled with words of S, whereas the leaf nodes are labeled with words of R. The idea is to place a separating word $v \in S$ at the root and partition all representatives $u \in R$ depending on whether $uv \in L$ or not: all representatives with $uv \notin L$ are put in the left subtree whereas all representatives with $uv \in L$ are put in the right subtree. This procedure is recursively repeated at each subtree (a separating word $v \in S$ might be used more than once) until all representatives are put in their own leaf node. In

this way, any two representatives $u \neq u' \in R$ are distinguished by the separating word of their least common ancestor in the tree. The algorithm labels the root node with $\varepsilon \in S$ to ensure that the representatives belonging to L are put in the root node's right subtree, whereas all other representatives are put in the left subtree. In the course of the learning process, the algorithm grows the tree preserving the properties described above.

From the classification tree t, Kearns and Vazirani's algorithm constructs a conjecture DFA $\mathcal{A}_t = (Q_t, \Sigma, q_0^t, \delta_t, F_t)$ in the following way. Representatives are used as states and, hence, the set of states is $Q_t = R$. Final states are those representatives $u \in R$ that are located in the right subtree of the root node (i.e., for which $u \cdot \varepsilon \in L$ holds). Moreover, the initial state is $q_0^t = \varepsilon$; remember that ε is always an element of R.

Kearns and Vazirani's algorithm determines transitions of $\mathcal{A}_t$ by using so-called *sifting operations*. Suppose that we want to define the transition $\delta_t(u, a)$ for some $u \in R$ and $a \in \Sigma$. A natural choice is a state (i.e., a representative) $u' \in R$ that has the same "behavior" (with respect to the classification tree) as the word ua. Such a representative can be found by *sifting ua down t*: starting at the root node, we descent at an inner node labeled with a separating word $v \in S$ to the left if $uav \notin L$ or to the right if $uav \in L$; we repeat this step recursively until we reach a leaf node. We denote the representative reached by sifting u down t by $\mathrm{sift}_t(u)$. The transitions of $\mathcal{A}_t$ are then defined by $\delta_t(u, a) = \mathrm{sift}_t(ua)$ where $u \in R$ and $a \in \Sigma$. Note that one can perform a sifting operation efficiently using membership queries, whose number depends on the height of the tree.

Algorithm 3.2 is a pseudo code implementation of Kearns and Vazirani's learning algorithm. The algorithm starts with a preprocessing step (Lines 1 to 9) in which it covers the cases $L = \emptyset$ and $L = \Sigma^*$. It first asks a membership query with ε. If the teacher replies $\varepsilon \in L$, it conjectures $L = \Sigma^*$ and asks an equivalence query with the one-state DFA $\mathcal{A}_{\Sigma^*}$ that accepts Σ^*. If the teacher replies $\varepsilon \notin L$, it conjectures $L = \emptyset$ and asks an equivalence query with the one-state DFA $\mathcal{A}_\emptyset$ that accepts the empty set. If the respective query passes, the algorithm halts and outputs the corresponding DFA.

If the teacher returns a counterexample w to the equivalence query, the algorithm sets $S = \{\varepsilon\}$ and $R = \{\varepsilon, w\}$. Moreover, it initializes the classification tree. The initial tree consists of a root node labeled with ε and two leaf nodes. If $\varepsilon \in L$, the left leaf node is labeled with w and the right one with ε. If $\varepsilon \notin L$, the left leaf node is labeled with ε and the right one with w.

Kearns and Vazirani's algorithm proceeds by asking an equivalence query with the DFA $\mathcal{A}_t$. As long as $|R| < \mathrm{index}(\sim_L)$, not all L-equivalence classes have been discovered. Thus, $L(\mathcal{A}_t)$ is necessarily different from L, and an equivalence query returns a counterexample. The algorithm uses this counterexample to identify a new representative and, thus, a currently unknown L-equivalence class.

Algorithm 3.2: Kearns and Vazirani's active learning algorithm.

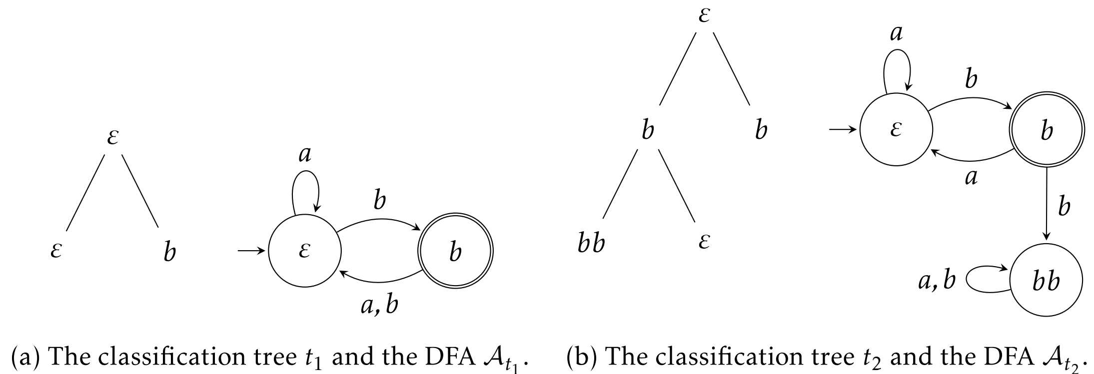
Input: A teacher for a regular language $L \subseteq \Sigma^*$.

1 Ask a membership query with ε.
2 **if** $\varepsilon \in L$ **then**
3 | Ask an equivalence query with the one-state DFA $\mathcal{A}_{\Sigma^*}$ with $L(\mathcal{A}_{\Sigma^*}) = \Sigma^*$.
4 **else**
5 | Ask an equivalence query with the one-state DFA $\mathcal{A}_\emptyset$ with $L(\mathcal{A}_\emptyset) = \emptyset$.
6 **end**
7 **if** *the teacher replies "yes" to the equivalence query* **then**
8 | **return** the corresponding DFA.
9 **end**
10 Set $S = \{\varepsilon\}$ and $R = \{\varepsilon, w\}$ where w is the counterexample of the equivalence query.
11 Initialize the classification tree t.
12 **repeat**
13 | Construct $\mathcal{A}_t$, and ask an equivalence query on $\mathcal{A}_t$.
14 | **if** *the teacher returns a counterexample* $w = a_1 \dots a_m$ **then**
15 | | Identify a breakpoint position $i \in \{1, \dots, m\}$.
16 | | $R \leftarrow R \cup \{u_{i-1}a_i\}$, $S \leftarrow S \cup \{a_{i+1} \dots a_m\}$.
17 | | Update t by splitting the leaf node labeled with u_i.
18 | **end**
19 **until** *the teacher replies "yes" to the equivalence query with* $\mathcal{A}_t$.
20 **return** $\mathcal{A}_t$.

Suppose that the teacher returns the counterexample $w = a_1 \dots a_m$. Kearns and Vazirani's algorithm now simulates the run $\mathcal{A}_t \colon u_0 \xrightarrow{a_1} u_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} u_m$ of $\mathcal{A}_t$ on w and searches for a so-called *breakpoint position* at which the conjecture is wrong when processing the remaining input. Formally, a breakpoint position is an index $i \in \{1, \dots, m\}$ that satisfies $u_{i-1}a_i \dots a_m \in L \Leftrightarrow u_i a_{i+1} \dots a_m \notin L$ where u_j is the representative that $\mathcal{A}_t$ reaches after reading $a_1 \dots a_j$; in particular, $u_0 = \varepsilon$. Note that a breakpoint position necessarily exists since w is a counterexample and $w = u_0 a_1 \dots a_m \in L \Leftrightarrow u_m \notin L$ holds.

A breakpoint position reveals that the words $u_{i-1}a_i$ and u_i belong to two different L-equivalence classes, which is witnessed by $a_{i+1} \dots a_m$. However, since $\delta_t(u_{i-1}, a_i) = u_i$, the word $u_{i-1}a_i$ is sifted into the leaf node labeled with u_i. This means that the representative u_i incorrectly represents two different L-equivalence classes, namely $[\![u_{i-1}a_i]\!]_L$ and $[\![u_i]\!]_L$. In fact, the word $u_{i-1}a_i$ is a new representative because it is already distinguished from all other representatives except from u_i (since $\mathrm{sift}_t(u_{i-1}a_i) = u_i$) but needs to be distinguished from u_i.

To reflect this new knowledge, the learning algorithm chooses a breakpoint position i (if there exist more than one, the choice is arbitrary). Then, it adds the new



(a) The classification tree t_1 and the DFA $\mathcal{A}_{t_1}$. (b) The classification tree t_2 and the DFA $\mathcal{A}_{t_2}$.

Figure 3.4: Classification trees and conjectures produced in Example 3.3.

representative $u_{i-1}a_i$ to R and the separating string $a_{i+1}\dots a_m$ to S (if it is not already contained). Moreover, it splits the leaf node labeled with u_i by replacing it with a subtree consisting of an inner node labeled with $a_{i+1}\dots a_m$ and two leaf nodes labeled with $u_{i-1}a_i$ and u_i; the order of these leaf nodes is determined by a membership query. Kearns and Vazirani's algorithm is then ready to produce the next conjecture.

Note that there can be several breakpoint positions for a single counterexample. Moreover, the same counterexample might be returned more than once during the learning process, thus, leading to different breakpoint positions. The correctness of Kearns and Vazirani's algorithm, however, does not depend on a particular choice of the breakpoint position.

The following example illustrates the functioning of Kearns and Vazirani's algorithm.

Example 3.3. To illustrate how Kearns and Vazirani's algorithm works, let us reconsider the target language $L_1 \subseteq \{a,b\}^*$ from Example 3.2 on Page 68.

In the preprocessing step, Algorithm 3.2 asks a membership query with ε, and the teacher replies $\varepsilon \notin L$. Thus, the algorithm asks an equivalence query with the DFA $\mathcal{A}_\emptyset$ that accepts the empty set. Since $L_1 \neq \emptyset$, the teacher returns a counterexample, say b.

Algorithm 3.2 now sets $S = \{\varepsilon\}$ and $R = \{\varepsilon, b\}$. Furthermore, it initializes the classification tree t_1 to be the tree depicted on the left hand side of Figure 3.4a.

Next, it constructs the DFA $\mathcal{A}_{t_1}$, which is shown on the right hand side of Figure 3.4a, and asks an equivalence query with this automaton. Since $L(\mathcal{A}_{t_1}) \neq L_1$, the teacher returns a counterexample, say bbb.

Algorithm 3.2 now searches for a breakpoint position $i \in \{1,2,3\}$. To this end, it simulates the run

$$\mathcal{A}_{t_1} : \varepsilon \xrightarrow{b} b \xrightarrow{b} \varepsilon \xrightarrow{b} b,$$

which yields $u_0 = u_2 = \varepsilon$ and $u_1 = u_3 = b$. Next, Algorithm 3.2 performs membership queries to identify a breakpoint position:

- For $i = 0$, it obtains $u_0 bbb = bbb \notin L$ and $u_1 bb = bbb \notin L$.
- For $i = 1$, it obtains $u_1 bb = bbb \notin L$ and $u_2 b = b \in L$.
- For $i = 2$, it obtains $u_2 b = b \in L$ and $u_3 \varepsilon = b \in L$.

Thus, $i = 2$ is the only breakpoint position, and Kearns and Vazirani's algorithm extends R with the new representative bb and S with the new separating word b. Furthermore, it replaces the leaf node labeled with ε with a subtree consisting of an inner node labeled with b and two leaf nodes labeled with ε and bb, respectively. The resulting classification tree t_2 is depicted on the left hand side of Figure 3.4b.

Finally, Algorithm 3.2 constructs $\mathcal{A}_{t_2}$, which is depicted on the right hand side of Figure 3.4b. Since $L(\mathcal{A}_{t_2}) = L_1$, the conjecture $\mathcal{A}_{t_2}$ passes an equivalence query and the learning terminates. ◁

The following theorem summarizes the key properties of Kearns and Vazirani's active learning algorithm.

Theorem 3.10 (Kearns and Vazirani [KV94], see also Balcázar et al. [BDGW97]). *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Algorithm 3.2 learns a DFA isomorphic to the canonical minimal DFA $\mathcal{A}_L$ in time polynomial in the size n of $\mathcal{A}_L$ and the length m of the longest counterexample returned by the teacher. It asks exactly n equivalence queries and $\mathcal{O}(n^2 + n \log_2 m)$ membership queries.*

Proof of Theorem 3.10. The cases $L = \emptyset$ and $L = \Sigma^*$ are handled in Lines 1 to 9. In both cases, Algorithm 3.2 asks one membership and one equivalence query. Moreover, it is not hard to see that Algorithm 3.2 returns a DFA isomorphic to $\mathcal{A}_L$ in these cases and that the effort to do so is constant.

We split the proof for all other cases into two parts. We first argue that the algorithm is correct (i.e., it terminates and returns a DFA that is isomorphic to $\mathcal{A}_L$). Then, we estimate the number of membership and equivalence queries.

Correctness We begin with two observations about the sets R and S and the classification tree t during the run of Algorithm 3.2.

1. Representatives are pairwise not L-equivalent. More formally, for any two representatives $u, u' \in R$ with $u \neq u'$ there exists an inner node of t labeled with $v \in S$ such that $uv \in L \Leftrightarrow u'v \notin L$. Hence, $u \not\sim_L u'$.
2. $\mathrm{sift}_t(u) = u$ holds for all representatives $u \in R$.

As long as $L(\mathcal{A}_t)$ is different from L, Algorithm 3.2 extends R with a new representative in every iteration of its main loop (Lines 12 to 19), thereby increasing $|R|$ by one. Moreover, $|R| \leq \text{index}(\sim_L)$ holds due to Observation 1. Thus, we need to show that $\mathcal{A}_t$ is isomorphic to $\mathcal{A}_L$ once $|R| = \text{index}(\sim_L)$.

To this end, suppose $|R| = \text{index}(\sim_L)$. The key idea is to prove that the classification tree then stores enough information about L to determine the L-equivalence classes of arbitrary words. More formally, we claim that then $w \sim_L \text{sift}_t(w)$ holds for all $w \in \Sigma^*$. Towards a contradiction, let $w \in \Sigma^*$ such that $w \not\sim_L \text{sift}_t(w)$. Due to Observation 1 and $|R| = \text{index}(\sim_L)$, there exists a representative $u \in R$ with $u \sim_L w$ and, hence, $u \neq \text{sift}_t(w)$. Additionally, since u and $\text{sift}_t(w)$ are two representatives with $u \neq \text{sift}_t(w)$ and due to Observation 2, there exists an inner node labeled with $v \in S$ such that $uv \in L \Leftrightarrow \text{sift}_t(w)v \notin L$. However, $\text{sift}_t(w)v \in L \Leftrightarrow wv \in L$ holds because w is sifted into $\text{sift}_t(w)$. This means that $w \not\sim_L u$, which is witnessed by v. This is a contradiction.

We are now set for showing that $\mathcal{A}_t$ is isomorphic to $\mathcal{A}_L$. To this end, we define the function $f \colon Q_t \to Q_L$ by $f(u) = [\![u]\!]_L$ where $u \in Q_t = R$ and claim that f is an isomorphism between $\mathcal{A}_t$ and $\mathcal{A}_L$. First, we note that f is injective due to Observation 1. Then, surjectivity of f immediately follows because $|Q_t| = \text{index}(\sim_L) = |Q_L|$ and f is injective.

It is left to show that f is a homomorphism (i.e., that it satisfies $f(\delta_t(u,a)) = \delta_L(f(u),a)$ for all $u \in R$ and $a \in \Sigma$). This is proven by the equations

$$f(\delta_t(u,a)) = f(\text{sift}_t(ua)) = [\![\text{sift}_t(ua)]\!]_L = [\![ua]\!]_L = \delta_L([\![u]\!]_L, a) = \delta_L(f(u), a),$$

where $[\![\text{sift}_t(ua)]\!]_L = [\![ua]\!]_L$ holds since $ua \sim_L \text{sift}_t(ua)$ and $[\![ua]\!]_L = \delta_L([\![u]\!]_L, a)$ holds by definition of $\mathcal{A}_L$ (see the definition of $\mathcal{A}_L$ on Page 14).

Complexity The main loop of Algorithm 3.2 repeats exactly $n-1$ times where $n = \text{index}(\sim_L)$. Due to the way Algorithm 3.2 extends the classification tree, the tree contains at most n leaf nodes, at most $n-1$ inner nodes, and its height is bounded by n. Hence, each sifting operation requires at most $n-1$ membership queries.

A naïve implementation of Kearns and Vazirani's algorithm constructs the DFA $\mathcal{A}_t$ from scratch in every iteration of the main loop. This requires $|R| \cdot |\Sigma| \leq n|\Sigma|$ sifting operations. Once the construction has been completed, the algorithm asks an equivalence query with $\mathcal{A}_t$. If the teacher returns a counterexample, it takes at most m membership queries to analyze the counterexample; remember that m is defined to be the length of the longest counterexample returned during the learning process. In summary, a naïve implementation of Algorithm 3.2 asks exactly n equivalence queries (one in its initialization phase and $n-1$ in its main loop) and $\mathcal{O}(n^3 + nm)$ membership queries.

Table 3.3: An overview of popular active learning algorithms with respect to their query complexity. The parameter n denotes the size of $\mathcal{A}_L$, and m denotes the length of the longest counterexample returned by the teacher.

Learning algorithm	Membership queries	Equivalence queries
Angluin's algorithm	$\mathcal{O}(n^2 m)$	$\leq n$
Kearns and Vazirani's algorithm	$\mathcal{O}(n^2 + n \log_2 m)$	n
Rivest and Schapire's algorithm	$\mathcal{O}(n^2 + n \log_2 m)$	$\leq n$

An improved version of Algorithm 3.2 caches membership queries that are asked during the construction of $\mathcal{A}_t$. Since representatives and separating words are added to the classification tree but never altered or deleted, it is indeed sufficient to pose a membership query with uav for $u \in R$, $a \in \Sigma$, and $v \in S$ only once. Moreover, a breakpoint position can be found more efficiently using a binary search. This improved version of Kearns Vazirani's algorithm also asks exactly n equivalence queries but requires only $\mathcal{O}(n^2 + n \log_2 m)$ membership queries.

Finally, let us note that all operations on the classification tree can be performed in time polynomial in the number of nodes of the tree. Furthermore, analyzing a counterexample can be done in time linear in the length of the counterexample. Thus, Algorithm 3.2 runs in time polynomial in m and n if implemented properly. □

3.2.3 Comparison

Besides Angluin's algorithm and Kearns and Vazirani's algorithm, there exists a third popular active learning algorithm due to Rivest and Schapire [RS93]. Rivest and Schapire's algorithm uses a *reduced* version of Angluin's observation table that stores exactly one representative per L-equivalence class. This has the advantage that the algorithm needs to store less data and asks less memberships queries. The improved observation table additionally removes the need for consistency checks since it guarantees that representatives are pairwise not L-equivalent. However, adding a counterexample to the table in the same way as done in Angluin's algorithm is no longer possible. Instead, Rivest and Schapire use the method based on finding a breakpoint position by means of a binary search as in Kearns and Vazirani's algorithm. (In fact, this method has originally been introduced by Schapire [Sch91].)

Table 3.3 compares the discussed active learning algorithms on the basis of their query complexity. Although Rivest and Schapire's algorithm is the superior algorithm from a theoretical perspective, it is worth pointing out two subtle, yet important points. First, the actual number of equivalence queries posed by Angluin's algorithm as well as Rivest and Schapire's algorithm can vary considerably depending on the target

language. In comparison to Rivest and Schapire's algorithm, Angluin's algorithm might require less equivalence queries because it potentially derives more information from counterexamples (as it adds counterexamples on the whole to the observation table). However, Angluin's way of processing counterexamples requires more membership queries. Second, Kearns and Vazirani's algorithm might need less membership queries than Rivest and Schapire's algorithm due to the more compact representation of the learned data; recall that Kearns and Vazirani's algorithm uses a classification tree, which distinguishes representatives using a subset of the separating words, whereas Rivest and Schapire's algorithm uses an observation table in which every combination of representative and separating word is stored. In summary, the choice of the learning algorithm should depend on how "expensive" answering equivalence queries in comparison to answering membership queries is. An appropriate choice can considerably improve the performance of the overall algorithm.

Let us conclude the section on active learning with a remark about the runtime of the discussed algorithms. We showed (e.g., in Theorems 3.9 and 3.10) that each algorithm runs in time polynomial in the size of the minimal DFA accepting the target language and the length of the longest counterexample returned by the teacher. In an application, however, it is clearly not enough to merely consider the runtime of the learning algorithm, but one also needs to take the time (and space) needed by the teacher to answer queries into account. In addition, the way how the teacher produces counterexamples influences the overall performance significantly. Thus, a meaningful complexity analysis has to be based on the combined runtime of the learner and the teacher.

However, in many applications (e.g., when the target language is not unique or when the target language is not regular) it is difficult to analyze the teacher's or the learner's exact complexity. In the first example, the lack of a unique target language does only allow to estimate the complexity with respect to the automaton eventually learned, which might not be related to the size of the input at all. In the second example, a learning algorithm is not guaranteed to terminate, which renders a complexity analysis useless. Therefore, we decided to omit a comprehensive theoretical analysis in such cases in favor of an efficiency proof by means of experimental evidence.

3.3 Libalf: the Automata Learning Framework

Libalf [BKK+10] is a comprehensive, open-source program library for learning finite automata. It was used for a large share of the experiments conducted in this thesis, and many of the learning algorithms used or developed in later chapters have been integrated into the library.

Table 3.4: Summary of learning algorithms implemented in Libalf.

	Algorithms
Active	Angluin's algorithm (two variants); the Angluin-style learner (see Section 4.3.4); the CEGAR-style learner (see Section 4.3.3); Kearns and Vazirani's algorithm; NL*; Rivest and Schapire's algorithm
Passive	Biermann and Feldman's algorithm (for passive learning without minimality constraint); Biermann and Feldman's method; DeLeTe2; Heule and Verwer's method; Grinchtein, Leucker, and Piterman's (unary and binary) methods; RPNI; the SMT-based method

The development of Libalf began with Kern's thesis [Ker09]. When Libalf was released in 2010, it comprised a total of nine learning algorithms, grouped into five active and four passive learning algorithms. At the time of writing, Libalf features seven active and eight passive learning algorithms. Table 3.4 compiles a full list.

Libalf unifies both active and passive learning techniques in a single easy-to-use and easy-to-extend library. The emphasis is on learning deterministic and nondeterministic finite automata but many algorithms are also capable of learning more complex automata models such as Moore machines. The library is written in C++ and designed for both research and application.

Libalf's key features are high flexibility and simple extensibility. High flexibility refers to the ability to let the user easily switch between learning algorithms and knowledge sources (often by changing only a single line of code). This allows one to easily experiment with different learning algorithms and to compare their assets and drawbacks in particular applications. In fact, all of Libalf's components are meant to be used in a plug-and-play manner, and no knowledge about the implementation is needed. Simple extensibility, on the other hand, refers to Libalf's generic structure, which allows developers to easily enrich Libalf with additional features, such as new learning algorithms, advanced automata models, domain-specific optimizations, and so on.

The library was designed to fit seamlessly into diverse environments. It runs on Microsoft Windows, Linux, and Mac OS (on both 32- and 64-bit architectures) and features a Java interface (which is based on the Java Native Interface). Additionally, Libalf provides a network-based client and server, which allows running Libalf remotely (e.g., on a high-performance machine).

Internally, Libalf represents words as lists of symbols where each symbol is an integer data type. Thus, the maximal size of an alphabet is 2^{32} or 2^{64} depending on the architecture of the target machine. Libalf uses a template mechanism to associate

words and their classifications, which makes it possible to use any C++ object as a classification of words.

The Libalf library has built-in support for several automata classes such as Mealy machines, Moore machines, as well as deterministic and nondeterministic finite automata. A Libalf automaton is derived from a generic yet simple interface, which makes adding new automata classes easy. Furthermore, Libalf provides an interface to the AMoRE finite automata library [KMP+89] should there be demand for a more powerful automaton library.

Libalf's main components are the learning algorithms and the so-called knowledgebase. The knowledgebase is an efficient, tree-like data structure to store words and their classifications. Each learning algorithm is associated with a knowledgebase, which serves as a buffer for the communication between the learning algorithm and the teacher. The gain of such external data storage is that the user is independent of the actual choice of the learning algorithm. In this way, it becomes possible to swiftly interchange different learning algorithms or to run them on the basis of the same data.

Libalf also features several domain-specific optimizations (which we do not want to discuss here) and auxiliary components to ease development and debugging. Most notable in this context are Libalf's adjustable logging facility, extensive usage, time, and memory statistics, and methods to generate GraphViz visualizations.

Finally, it is worth noting that Libalf was not only used in Kern's and this thesis but also proved its usefulness in various other works. To name but a few, Libalf is used for learning workflow petri nets [ELS11], for the analysis of probabilistic systems [FKP10, FKP11], and also the next generation learnlib [MSHM11] offers an interface to Libalf.

4

Regular Model Checking

Our first application scenario for automata learning in the context of verification is Regular Model Checking. Regular Model Checking is a special type of Model Checking[1] in which the program—or system—at hand is modeled symbolically in terms of finite automata. More precisely, configurations of a program are modeled as finite words, sets of configurations as regular languages, and the program's semantic (i.e., its transitions) as a rational relation over pairs of configurations. A key motivation for modeling programs in terms of regular languages is that (deterministic) finite automata provide an efficient representation of large and infinite state spaces.

We here consider the verification of safety properties as in the case of the original Regular Model Checking framework [KMM⁺97]. More precisely, we want to verify that a given program does not allow for an execution that starts in a set I of *initial configurations* and ends in a dedicated set B of *bad configurations*; the bad configurations describe conditions of the program that must not occur during its execution. However, this question is undecidable in general and tools for Regular Model Checking are, therefore, necessarily *incomplete* (i.e., they give the correct answer on termination but are not guaranteed to halt). Nonetheless, good results of such algorithms were reported for a large number of practical applications [Leg08, BFL04].

In general, verification techniques can broadly be classified into two categories: *white-box* techniques, which have complete access to the internals of the program in question, and *black-box* techniques, which are (largely) agnostic of the program and typically obtain their information from an external source.

The majority of existing tools and techniques for Regular Model Checking, such as Faster [BFL04] and T(o)rmc [Leg08], fall in the first category as they directly

[1] We assume a familiarity with the fundamentals of Model Checking and refer the reader to Baier and Katoen [BK08] for an introduction to this topic.

manipulate the input-automata. A general advantage of white-box techniques is that they can use every bit of information available to reason about the given program. At the same time, however, complete knowledge of a (potentially complex) program can also be disadvantageous, and building a verification procedure that works cognizant of the program is hard, simply due to the fact that one has to handle the complex logic of the program. In the context of Regular Model Checking, for instance, the size of the input-automata (which provides a natural measure for the complexity of the program) is a prime factor for the actual performance of algorithms. In fact, the applicability of tools such as Faster and T(o)rmc is often seriously limited in practical applications by the size of the input-automata.

Black-box techniques, on the other hand, offer an elegant solution to this problem as they avoid working directly on the given automata. Automata learning seem particularly useful in this context: since such techniques are agnostic to the complexity of the program at hand and obtain their information from sample executions or from an external information source, they typically produce simple solutions. Another promising feature is that automata learning algorithms are good at generalizing from few information, whereas white-box algorithms often struggle with the comprehensive knowledge they posses about the program. Moreover, the complexity of learning-based algorithms usually depends on the final result but not immediately on the input. Therefore, developing learning-based black-box techniques for Regular Model Checking seems worthwhile.

The objective of this chapter is to study how automata learning can advance the state-of-the-art in Regular Model Checking. To this end, we develop several learning-based algorithms and contrast them with existing tools. Our algorithms are based upon an approach commonly used to verify safety properties: we search for an *invariant* of the program that contains at least the reachable configurations and is disjoint from the set of bad configurations. More precisely, we aim for a set Inv of program configurations that

- contains all of the initial configurations in I;
- contains none of the bad configurations in B; and
- is *inductive* (i.e., $c \in Inv$ implies $c' \in Inv$ for all transitions (c, c') of the program).

Such a set is in fact enough to prove a program correct because it witnesses that there is no way to reach a bad configuration from an initial configuration. Since program configurations are modeled as finite words, we use DFAs as representations of invariants. We introduce all necessary formalisms and notations in Section 4.1.

The pivotal idea of the algorithms presented next is to separate the program to verify from the actual invariant synthesis algorithm and use Angluin's learning framework

as a standard protocol for data exchange: a membership query corresponds to the question whether a configuration is to be included in an invariant, whereas an equivalence query corresponds to the question whether a conjecture accepts an invariant. In the context of Regular Model Checking, however, we cannot simply apply standard learning algorithms because we cannot build a teacher who has precise knowledge about an invariant (as this would entail solving the Regular Model Checking problem beforehand). The situation becomes even more intricate because there might exist several valid invariants, which means that the teacher's target concept is no longer unique. Indeed, the lack of a precise teacher for invariants is a great challenge, and we have to carefully design new learning settings to be able to apply automata learning techniques.

In the course of this chapter, we develop a family of three different types of algorithms for computing invariants in Regular Model Checking. In Section 4.2, we start with a white-box algorithm. Although this algorithm is not yet based on automata learning, it serves as basis for the following learning-based algorithms. On top of that, it is also of its own interest in that it avoids many difficulties that existing white-box techniques have to deal with. Subsequently, in Section 4.3, we develop two algorithms that combine active and passive learning. These algorithms obtain their information about the initial and bad configurations from a teacher, but they still need access to the transducer of the program. Thus, one might think of these algorithms as halfway between white-box and black-box approaches. Finally, in Section 4.4, we modify the algorithms of Section 4.3 to completely operate in a black-box fashion. The algorithms of Section 4.4 obtain all of their information solely from a teacher, whose task it is to reason about the program. Thus, these algorithms can also be applied in situations in which the program cannot be modeled in terms of regular languages, the program is given as a black-box, or the input-automata are too large for white-box techniques.

The remainder of this introduction gives an overview of each type of algorithm and concludes with a summary of topics also covered in this chapter.

A white-box algorithm Our first algorithm works in a complete white-box fashion: it takes two NFAs accepting the sets of initial and bad configurations as well as an asynchronous transducer as input and produces a DFA accepting an invariant as output. Internally, the algorithm constructs logic formulas that postulate the existence of such a DFA and delegates their satisfiability checks to an underlying logic solver. More precisely, our algorithm creates and solves a sequence of logic formulas φ_n where $n \in \mathbb{N}_+$ that depend on the input automata and have the following two properties: first, φ_n is satisfiable if and only if there exists a DFA with n states that accepts an invariant; second, a model of φ_n acts as a blueprint to construct such a DFA. Starting with $n = 1$, the algorithm successively increases n until φ_n becomes satisfiable. This procedure

guarantees to find a smallest DFA accepting an invariant (in terms of the number of states) provided that one exists.

We implement the formulas φ_n in two different logics, which already proved their efficacy in the context of passive learning (cf. Section 3.1): Propositional Boolean Logic and the logic of uninterpreted functions over the naturals. Since mature solvers for both logics are available, the hope is—and we substantiate this hope experimentally—that a solver-based algorithm also proves its effectiveness in the context of Regular Model Checking.

Although the algorithm sketched above is white-box and does not use automata learning, it still offers an advantage over existing tools in that it avoids manipulating the input-automata. Broadly speaking, tools such as T(o)rmc and Faster create a sequence of DFAs, starting with a DFA accepting the set of initial configurations, by applying the transition relation until a DFA accepts an inductive set that is disjoint from the set of bad configuration. Both tools use abstraction techniques during their computation (*acceleration* in the case of Faster and *extrapolation* in the case of T(o)rmc) in order to make the sequence of DFAs converge in finite time.[2] However, a serious drawback of such approaches is that they often produce huge intermediate results (although T(o)rmc tries to reduce their size during the computation), even if their final result itself is small. In contrast, our solver-based algorithm uses a highly optimized logic solver to handle expensive calculations (our experiments show that the used solvers performed well on the resulting formulas). Moreover, our algorithm is deliberately designed to search for a smallest invariant, which helps to keep the logic formulas and the effort to solve them as small as possible.

Semi-black-box algorithms Our semi-black-box algorithms learn an invariant in interaction with a teacher. The underlying idea is to abstract from the exact sets of initial and bad configurations by sampling them. More precisely, our algorithms maintain a sample $\mathcal{S} = (S_+, S_-)$ that consists of a finite set S_+ approximating the initial configurations and a finite set S_- approximating the bad configurations. In every iteration, the algorithms compute a DFA that is consistent with $\mathcal{S}$ and inductive with respect to the transducer by means of a logic solver (which involves encoding the transducer into the logic formula as in the case of the white-box algorithm). The resulting DFA contains at least the configurations reachable from S_+ via the transitions defined by the transducer and does not contain any configuration in S_-. If all initial configurations and no bad configurations are contained, the DFA accepts an invariant. If this is not the case, the respective approximation needs to be refined.

[2]We describe the functioning of both Faster and T(o)rmc in more detail in the section about related work.

The actual learning takes place in an Angluin-style active learning setting. More precisely, we assume a setting in which the teacher answers queries as described next.

Membership query On a membership query, the teacher returns whether the configuration in question is an initial configuration (the teacher returns “yes”) or whether it is a bad configuration (the teacher returns “no”). If the configuration is neither initial nor bad, the teacher does not know whether an invariant includes or excludes the configuration (as the problem is learning an unknown invariant) and answers “don’t know”.

Equivalence query On an equivalence query, the teacher only permits conjectures that accept an inductive set. Then, checking whether the conjecture accepts an invariant amounts to checking whether it classifies the sets of initial and bad configurations correctly. Should that not be the case, the teacher can easily identify a counterexample.

We call such teachers *incomplete* because they can only provide incomplete information about an invariant.

We propose two learners capable of learning from incomplete teachers (i.e., learners that can handle “don’t cares” and produce conjectures accepting inductive sets), which differ in the strategy to sample and refine sets of program configurations. The first learner follows the idea of the *CEGAR framework* [CGJ+00]: if the abstraction of either the initial or the bad configurations is too coarse and a conjecture does not accept an invariant, the teacher returns a counterexample and the learner refines the abstraction accordingly. The second learner follows a more elaborated procedure based on Angluin’s learning algorithm, where additional membership queries ask whether individual configurations belong to an invariant. These queries refine the abstraction further and remove the need of generating a new conjecture in every step.

To complete our semi-black-box approach, we describe exemplary how to construct an incomplete teacher, assuming that the program at hand is given in terms of finite automata. Note, however, that the semi-black-box setting does not prescribe in which form the sets of initial and bad configurations have to be given but only requires an incomplete teacher to work.

Black-box algorithms Finally, we turn the incomplete teacher setting from above into an Angluin-style black-box learning setting. The learner is now completely agnostic of the program being verified and obtains all information exclusively from a teacher, who reasons about the program. Therefore, the learner’s objective becomes to build a conjecture that satisfies the teacher’s demands.

However, this learning setting comes with an inherent problem. Consider an equivalence query with a DFA accepting a set P of program configurations; if P is not

inductive, say due to a transition (c, c') with $c \in P$ and $c' \notin P$, then the teacher is stuck: since he does not know an invariant, the teacher has no information about whether c' should be included in or c should be excluded from a conjecture. In current state-of-the-art learning algorithms for invariant synthesis [CGP03, AMN05a, ACMN05], the teacher cheats: whenever a conjecture is not inductive, he makes an arbitrary choice in the hope that this choice will still result in an invariant. However, this makes the setting nonrobust, "causing divergence, blocking the learner from learning the simplest concepts, and introducing arbitrary bias that is very hard to control" [GLMN13a, Page 3].

As solution to this problem, we recently proposed the so-called *ICE-learning framework (learning via implications, counterexamples, and examples)* [GLMN13a, GLMN14], which extends Angluin's learning setting with *implication counterexamples*: if the DFA conjectured on an equivalence query accepts a noninductive set P, the teacher returns a pair of program configurations witnessing a violation of inductivity (i.e., he returns a pair (c, c') such that $c \in P$, $c' \notin P$, and c' is reachable from c via a transition of the program). This way, a teacher can precisely communicate why a conjecture is not inductive even without knowing an invariant.

The exact learning setting we use for Regular Model Checking is a combination of the incomplete teacher setting and the ICE-learning framework. A teacher for this black-box setting answers queries as follows.

Membership query On a membership query, the teacher works in the same way as the incomplete teacher and returns "yes", "no", or "don't know".

Equivalence query Equivalence queries are no longer restricted. Given a conjecture, the teacher returns "yes" if it accepts an invariant, a (classical) counterexample if it classifies the initial or bad configurations incorrectly, or an implication-counterexample if the conjecture's language is not inductive.

This new learning setting requires us to construct a learning algorithm that can handle both "don't know" answers and implication-counterexamples. To this end, we adapt our semi-black-box algorithms to encode implications rather than the program's transducer in the logic formulas. Moreover, we describe exemplary how to build an appropriate teacher, again assuming that the program at hand is given in terms of finite automata. As in the case of the semi-black-box setting, the black-box setting does not prescribe in which form the program has to be given but only requires an ICE-teacher to work.

Evaluation and further applications Based on a prototype implementation, we compare our algorithms with Faster and T(o)rmc. To this end, we use two benchmark

suites: the first suite originates from the Faster website and contains implementations of numerous protocols; the second suite contains implementations of the well-known token ring protocol and a modulo counter. A feature of the latter suite is that one can easily vary the size of the input-automata, which we use to demonstrate the advantage of our black-box algorithms when confronted with large programs. We present and discuss the experimental results in Section 4.5.

In Section 4.6, we apply our algorithms to two application scenarios beyond the scope of Regular Model Checking. The first application is *minimal separating DFAs*, which play an important role, for instance, in the context of compositional verification [CFC+09]. The second application is automatic loop invariant synthesis in the context of Floyd-Hoare-style verification of while programs.

We conclude this chapter in Section 4.7.

Parts of this chapter, in particular Sections 4.2 and 4.6, appeared in conference proceedings [Nei12]. The idea of learning invariants from incomplete teachers is based on joint work with Martin Leucker [LN12], in which a similar scenario called learning from inexperienced teacher is discussed. The semi-black-box algorithm of Section 4.3 and the experiments of Section 4.5 are joint work with Nils Jansen [NJ13]. The ICE-learning paradigm of Section 4.4 is joint work with Pranav Garg, Christof Löding, and P. Madhusudan [GLMN13a, GLMN14].

Related Work

Regular Model Checking has been introduced by Kesten et al. [KMM+97]. In their work, the authors are interested in verifying safety properties and assume that the transitions of a program are given in terms of a synchronous transducer. To prove a program correct, they describe a simple symbolic fixed-point algorithm that iteratively applies the transducer until the set of reachable configurations is computed; once this has been done, a simple inclusion check reveals whether a bad configuration is reachable. For infinite-state systems, however, this algorithm clearly is not guaranteed to terminate.

Bouajjani et al. [BJNT00] were among the first to adopt the Regular Model Checking paradigm. The authors developed a more elaborate algorithm that computes a synchronous transducer accepting the reflexive and transitive closure of the transition relation by using *quotienting* to make the process converge in finite time (if the input permits this). Moreover, they identified syntactic constraints on the input that guarantee that the desired transducer can be computed by their algorithm. These constraints, however, are rather restrictive.

The survey of Abdulla et al. [AJNS04] provides a comprehensive overview of Regular Model Checking techniques. This survey includes techniques based on quotienting [BJNT00, DLS01], *abstraction* [BHV04], and *extrapolation* [BLW03, Tou01].

An example of a learning-based approach to Regular Model Checking has been introduced by Habermehl and Vojnar [HV05]. Their idea is to sample all words of a fixed length $n \in \mathbb{N}_+$ and to use a passive learning algorithm (in their case the Trakhtenbrot-Barzdin algorithm [TB73]) in order to obtain a DFA that accepts an invariant; if this does not result in an appropriate DFA, the parameter n is increased and the procedure is repeated. Although Habermehl and Vojnar's algorithm is guaranteed to find a DFA accepting an invariant if one exists, a major drawback of this technique is that the sample grows exponentially in n.

Apart from the approaches mentioned above, we are aware of two mature tools for Regular Model Checking that proved to be successful in practice: T(O)RMC [Leg08], and LEVER [VV06]. FAST [BFLP03] and the successor FASTER [BLP06] are also often mentioned in the context of Regular Model Checking, although these tools solve a slightly different task. For the reader's convenience, let us briefly introduce these tools.

T(O)RMC T(O)RMC is a tool for the Regular Model Checking problem that is based on extrapolation. Given a deterministic asynchronous transducer[3] $\mathcal{T}$ and a DFA $\mathcal{A}^I$, T(O)RMC aims at computing a DFA accepting the set $R(\mathcal{T})^*(\mathcal{A}^I)$, which amounts to computing the limit of the sequence $R(\mathcal{T})^0(\mathcal{A}^I), R(\mathcal{T})^1(\mathcal{A}^I), \ldots$ To this end, T(O)RMC first computes a finite sequence $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ of minimal DFAs that satisfy $L(\mathcal{A}_i) = R(\mathcal{T})^i(\mathcal{A}^I)$ by iterating $\mathcal{T}$ (cf. Page 16). Then, it heuristically picks a so-called *sampling sequence* $0 \leq i_1 < i_2 < \cdots < i_m \leq n$ and searches for differences in the transition structures of the DFAs $\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \ldots, \mathcal{A}_{i_m}$ in the form of *increments* (see Legay [Leg07] for details). Once T(O)RMC has identified an increment, it extrapolates the repetition of this increment by adding loops to the last automaton of the sampling sequence. After doing so, T(O)RMC heuristically checks whether the extrapolated limit is exact. If this is the case, T(O)RMC outputs the resulting DFA, and the user can check the result for an empty intersection with the set of bad configurations. If T(O)RMC did not succeed in computing a DFA accepting $R(\mathcal{T})^*(\mathcal{A}^I)$, the user needs to choose another policy for picking the sampling sequence and has to restart the whole procedure.

It is worth mentioning that T(O)RMC is not only a tool for Regular Model Checking but also for *ω-Regular Model Checking* [BLW04]. There, configurations are modeled as infinite words (rather than finite words) and sets of configurations are defined in terms of deterministic weak Büchi automata.

[3] An (a)synchronous transducer $\mathcal{T}$ is called *deterministic* if there exists at most one run of $\mathcal{T}$ on every input.

Lever Lever is a tool for verifying infinite-state systems that has been developed in the context of the *Learning to Verify Project* [VSVA04a, VSVA04b, VSVA05, VV05]. It is based on automata learning in an active Angluin setting and uses a modified version of Kearns and Vazirani's learning algorithm. Besides Regular Model Checking, Lever also supports the verification of liveness properties.

Lever learns the unique fixed point of a suitable functional that depends on the property to verify. This fixed point is a set of so-called *configuration-witness pairs*, which consist of a configuration c and a witness for the fact that c indeed belongs to the fixed point. In the case of Regular Model Checking, a natural number i witnessing that c is reachable from some initial configuration by traversing at most i transitions is sufficient for this purpose.

When used for Regular Model Checking, Lever tries to learn the (minimal) DFA accepting the set of configuration-witness pairs and obtains the set of reachable configurations simply by discarding the witness component. An answer to the Regular Model Checking problem can then be given by checking whether the resulting DFA accepts a bad configuration. The fact that the target concept is unique allows applying standard active learning techniques.

Although the authors report good results on various test cases, Lever is no longer publicly available. According to one of the authors, the development and maintenance has stopped.

Fast and Faster Fast and Faster were not designed for Regular Model Checking but for verifying safety properties of counter systems that are modeled as automata augmented with unbounded integer variables. More precisely, the input of Fast and Faster consists of the following:

- A counter system, given as an automaton whose transitions are labeled with Presburger formulas that specify guards and describe the effect of a transition on the variables
- A set of initial configurations, given as a Presburger formula
- Optionally, a set of bad configurations, also given as a Presburger formula

Due to the connection between Presburger formulas and finite automata (cf. Section 2.2), inputs for Fast and Faster can be expressed as Regular Model Checking instances, and many Regular Model Checking problems can be translated into inputs for Fast and Faster. This makes it often possible to apply Fast and Faster in the context of Regular Model Checking.

The goal of Fast and Faster is to compute the exact set of reachable configurations and, if desired, to check whether this set contains a bad configuration. Faster is an

extension of the predecessor Fast, which uses the same underlying paradigms and improves Fast, among other things, with an interface to popular automata libraries. Both tools are based on *acceleration techniques*. Similar to T(o)rmc, Fast and Faster compute a sequence of DFAs that serves as basis for identifying so-called *cycles* (see Bardin, Finkel, and Leroux [BFL04] for a definition of cycles). Once a “good” cycle has been identified (according to some heuristic measures), Fast and Faster compute the acceleration relation in the sense of Finkel and Leroux [FL02] and use it to derive a DFA that corresponds to the in-the-limit effect of iterating this cycle. This procedure is repeated until all good cycles have been found. If bad configurations are specified, the tools finally check for an empty intersection with the resulting DFA and report the answer.

Extensions of Regular Model Checking It is worth mentioning that there exist several variations and extensions of Regular Model Checking. These can broadly be grouped along two dimension.

The first dimension refers to the formalism that is used to describe the program at hand. Perhaps most notably, Regular Model Checking has been lifted to infinite words [Leg08, BLW04] and (finite) trees [AJMd02, BHRV12]. Moreover, Fisman and Pnueli [FP01] describe a symbolic Model Checking technique in which the program is modeled using a combination of regular languages and a particular subclass of deterministic context free languages.

The second dimension refers to the type of properties to be verified. For instance, Regular Model Checking has been applied to the verification of liveness and certain other ω-regular properties (see Vardhan and others [VV05, VSVA05]).

4.1 Regular Model Checking and Invariants

In the Regular Model Checking framework, configurations of a program are modeled as finite words over an a priori fixed alphabet Σ. The *program* itself consists of a set of initial configurations and the transitions. In the following, we consider a version in which the transitions are a rational relation (defined by an asynchronous transducer).

Definition 4.1 (Program)**.** Given an alphabet Σ, a *program* is a tuple $\mathcal{P} = (I, T)$ consisting of a regular set $I \subseteq \Sigma^*$ of *initial configurations* and a rational *transition relation* $T \subseteq \Sigma^* \times \Sigma^*$ (i.e., the successor relation on the configurations).

Regular Model Checking—more precisely, the *Regular Model Checking problem*—is the decision problem

"Given a program $\mathcal{P} = (I, T)$ and a regular set $B \subseteq \Sigma^*$ of *bad configurations* with $I \cap B = \emptyset$. Does $T^*(I) \cap B = \emptyset$ hold?"[4]

The program $\mathcal{P}$ induces a rational graph $G_{\mathcal{P}} = (\Sigma^*, T)$ whose vertices are configurations and whose edges are transitions. Based on this view, the Regular Model Checking problem asks whether a path in $G_{\mathcal{P}}$ exists that leads from some initial configuration into the set of bad configurations. If such a path exists, the program is erroneous and the sequence of transitions along this path represents a spurious execution of the program. Note that we here require I and B to be disjoint; if this is not the case, one can immediately answer the Regular Model Checking problem negatively.

Let us illustrate Regular Model Checking with an example.

Example 4.1. Let us consider the token ring protocol, used as illustrative example by Bouajjani et al. [BJNT00]. In this example, the program $\mathcal{P}$ models an arbitrary but finite number of processes arranged in a token-ring topology, which pass a token from one process to the next. The set of configurations of this program is the set of all words over $\Sigma = \{0, 1\}$: the length of a word corresponds to the number of processes, and the i-th symbol of a word indicates whether the i-th process currently possesses a token.

The automata in Figure 4.1 (on Page 92) model the token ring protocol. The set I, accepted by the NFA $\mathcal{A}^I$ in Figure 4.1a, contains all words of the form 10^*, which model that initially the first process is in possession the only token. The set B, accepted by the NFA $\mathcal{A}^B$ in Figure 4.1b, accepts all words with either none or at least two occurrences of 1, which models the requirement that a token must neither be lost nor multiplied during the program's execution. The transitions of the program are defined by the asynchronous transducer $\mathcal{T}$ in Figure 4.1c, which accepts words of the form $(0,0)^*(1,0)(0,1)(0,0)^*$ or $(1,0)(0,0)^*(0,1)$. Words of the first form model situations in which Process i passes the token to Process $i+1$ whereas words of the latter form model situations in which Process n passes the token to Process 1. ◁

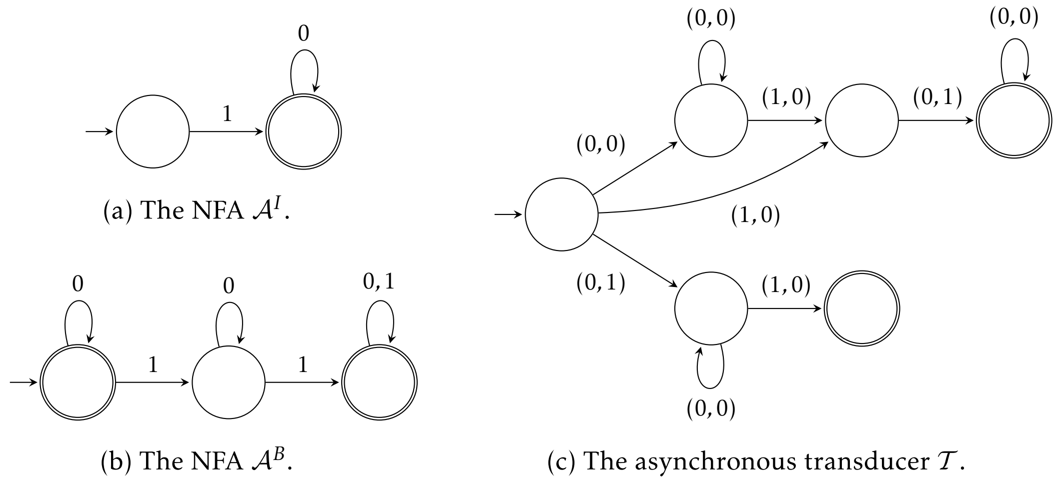
The Regular Model Checking problem straightforwardly translates into a non-reachability problem over rational graphs. Since the reachability problem for rational graphs is undecidable (see Section 2.1) and decidability of a problem implies decidability of the problem's complement, the following remark is immediate.

Remark 4.1. The Regular Model Checking problem is undecidable.

Remark 4.1 implies in particular that all decision procedures for the Regular Model Checking problem can be *incomplete* at best; that is, such algorithms give the correct answer on termination but are not guaranteed to halt.

Although the Regular Model Checking problem is undecidable, various methods have been described in the literature to approach the problem (again, we refer the

[4] Recall that T^* denotes the reflexive and transitive closure of T and $T(I)$ the image of I under T.



(a) The NFA $\mathcal{A}^I$.

(b) The NFA $\mathcal{A}^B$.

(c) The asynchronous transducer $\mathcal{T}$.

Figure 4.1: Automata modeling the correctness of the token ring protocol as a Regular Model Checking problem.

reader to Abdulla et al. [AJNS04] for a comprehensive overview). We here follow a popular approach and search for a so-called invariant of the program at hand.

Definition 4.2 (Invariant)**.** An *invariant* of a program $\mathcal{P} = (I, T)$ over the alphabet Σ with respect to a set $B \subseteq \Sigma^*$ of bad configurations is a set $Inv \subseteq \Sigma^*$ that satisfies

1. $I \subseteq Inv$;
2. $B \cap Inv = \emptyset$; and
3. $c \in Inv$ implies $c' \in Inv$ for all $(c, c') \in T$.

A set satisfying Condition 3 is called *inductive with respect to T*—or closed under the program's transitions. Moreover, we call a set satisfying Definition 4.2 simply an invariant if $\mathcal{P}$ and B are clear from the context.

Broadly speaking, an invariant is an overapproximation of the set of reachable configurations that is inductive and has an empty intersection with the set of bad configurations. Given an invariant *Inv*, a simple induction shows that any execution starting in a configuration in *Inv* stays in *Inv* and, hence, never encounters a bad configuration. Thus, an invariant is indeed enough to prove a program correct.

Searching for an invariant rather than computing the set of reachable configurations, or the transitive closure of the transition relation for that matter, has advantages: for one thing, an invariant might be computable even if the set the of reachable configurations is not; for another, since an invariant (over-)approximates the set of reachable configurations, it often has a much simpler structure.

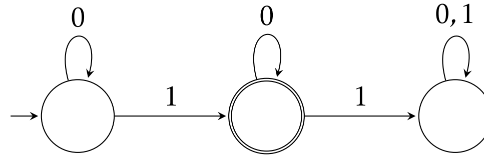


Figure 4.2: An IDFA for the Regular Model Checking problem of Example 4.1.

We use DFAs as finite representations of invariants. To ease description, we lift the notion of inductiveness to DFAs and introduce *invariant-DFAs*. This kind of DFAs is what our algorithms aim for.

Definition 4.3 (Inductive DFA, Invariant-DFA)**.** Let $\mathcal{P} = (I, T)$ be a program over an alphabet Σ and $B \subseteq \Sigma^*$ a set of bad configurations. Moreover, let $\mathcal{T}$ be an asynchronous transducer.

- A DFA $\mathcal{A}$ is called *inductive with respect to* T if $L(\mathcal{A})$ is inductive with respect to T; analogously, $\mathcal{A}$ is called *inductive with respect to* $\mathcal{T}$ if $L(\mathcal{A})$ is inductive with respect to $R(\mathcal{T})$. If T, respectively $\mathcal{T}$, is clear from the context, we simply say that $\mathcal{A}$ is inductive.
- A DFA is called an *invariant-DFA (IDFA)* if $L(\mathcal{A})$ is an invariant with respect to $\mathcal{P}$ and B. Again, we omit references to $\mathcal{P}$ and B if they are clear from the context.

Let us illustrate Definition 4.3 with an example.

Example 4.2. Reconsider the token ring protocol of Example 4.1 (on Page 91). The DFA depicted in Figure 4.2 is an IDFA: since it accepts all words that contain the symbol 1 exactly once, it accepts all configurations in I and rejects all in B; moreover, it is not hard to verify that the DFA is also inductive. ◃

Note that this choice of DFAs as representation of invariants is a restriction since there might be invariants that cannot be recognized by a DFA. Nonetheless, almost all techniques for Regular Model Checking follow this approach, mainly due to the fact that DFAs enjoy good closure properties and allow manipulating infinite sets of program configurations effectively.

For the remainder of this chapter, we fix a program $\mathcal{P} = (I, T)$ over an alphabet Σ together with a set $B \subseteq \Sigma^*$ of bad states. Furthermore, we assume (if not explicitly stated otherwise) that an IDFA exists; this implies $I \cap B = \emptyset$ in particular.

4.2 A White-box Algorithm

Our first algorithm for computing invariants (in form of IDFAs) is a white-box algorithm that expects the program given by

- an NFA $\mathcal{A}^I = (Q_I, \Sigma, q_0^I, \Delta_I, F_I)$ accepting the set I of initial configurations;
- an NFA $\mathcal{A}^B = (Q_B, \Sigma, q_0^B, \Delta_B, F_B)$ accepting the set B of bad configurations; and
- an asynchronous transducer $\mathcal{T} = (Q_T, \Sigma, \Sigma, q_0^T, \Delta_T, F_T)$ defining the transitions T of the program.

The key idea of our algorithm is to search for an IDFA by constructing and solving a sequence of logic formulas $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$ for increasing values of $n \in \mathbb{N}_+$. These formulas depend on $\mathcal{A}^I$, $\mathcal{A}^B$ as well as $\mathcal{T}$ and are designed to have the following two properties:

1. The formula $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$ is satisfiable if and only if there exists an IDFA $\mathcal{A}$ with n states; that is, $\mathcal{A}$ satisfies $L(\mathcal{A}^I) \subseteq L(\mathcal{A})$, $L(\mathcal{A}^B) \cap L(\mathcal{A}) = \emptyset$, and $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$.
2. If $\mathfrak{M}$ is a model of $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$, then $\mathfrak{M}$ contains enough information to construct an IDFA $\mathcal{A}_{\mathfrak{M}}$ with n states.

Clearly, if $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$ is satisfiable for some value of n, then Property 2 guarantees that we can construct an IDFA from a model of $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$. However, if the formula is unsatisfiable but an IDFA exists, then the parameter n has been chosen too small and we increment it. It is not hard to verify that this process indeed terminates eventually if an IDFA exists. Algorithm 4.1 describes this procedure in pseudo code.

Algorithm 4.1: Computing IDFAs using logic solver.

Input: Two NFAs $\mathcal{A}^I$, $\mathcal{A}^B$ and an asynchronous transducer $\mathcal{T}$.

```
1 n ← 0.
2 repeat
3 |  n ← n + 1.
4 |  Construct and solve φ_n^{A^I, A^B, T}.
5 until φ_n^{A^I, A^B, T} is satisfiable with model 𝔐.
6 return the IDFA A_𝔐 constructed from 𝔐.
```

An alternative approach is to use a binary search and halt once an IDFA has been found rather than incrementing n by one in each iteration. However, increasing n stepwise by one keeps the formula $\varphi_n^{\mathcal{A}^I, \mathcal{A}^B, \mathcal{T}}$ as small as possible and avoids large intermediate results.

We can now state the main result of this section.

Theorem 4.1. *Let $\mathcal{P} = (I, T)$ be a program given as NFA $\mathcal{A}^I$ and asynchronous transducer $\mathcal{T}$ over the common alphabet Σ, and let $B \subseteq \Sigma^*$ be a regular set of bad configurations given as NFA $\mathcal{A}^B$. If an IDFA with respect to $\mathcal{P}$ and B exists, say with k states, then Algorithm 4.1 terminates after at most k iterations and returns a smallest IDFA.*

Proof of Theorem 4.1. The proof of Theorem 4.1 is a straightforward application of the fact that the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ has indeed the desired properties (see Lemmas 4.5 and 4.6 on Page 99 and Page 101, receptively): if an IDFA with k states exists, then $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ is satisfiable for all $n \geq k$ and we can use a model of the formula to construct an IDFA. Moreover, increasing n by one in every iteration of the loop guarantees that one finds a smallest IDFA with respect to the number of states in at most k iterations. □

In the following two subsections, we use two different logics to implement the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$, which were already successfully used in the context of passive learning. The first is Propositional Boolean logic, which we consider in Section 4.2.1. In Section 4.2.2, we consider the quantifier-free logic of uninterpreted functions over the naturals.

4.2.1 Computing IDFAs Using Propositional Boolean Logic

We now develop a formula in Propositional Boolean Logic that, if satisfiable, encodes an IDFA with a fixed number $n \geq 1$ of states. For this purpose, we fix the set of states to $Q = \{q_0, \dots, q_{n-1}\}$ and the initial state to $q_0 \in Q$. To encode IDFAs in Propositional Boolean Logic, we follow Heule and Verwer's idea for passive learning (see Section 3.1.4), which is based on the following simple observation: if the set of states and the initial state are fixed (e.g., as above), then each DFA is completely determined by its transition function and final states. We exploit this fact and use Boolean variables $d_{p,a,q}$ and f_q where $p, q \in Q$ and $a \in \Sigma$. Assigning *true* to $d_{p,a,q}$ means that the transition $\delta(p, a) = q$ exists in the prospective DFA. Similarly, assigning *true* to f_q means that q is a final state.

To make sure that the variables $d_{p,a,q}$ indeed encode a deterministic transition function, we impose the following constraints:

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{q,q' \in Q,\ q \neq q'} \neg d_{p,a,q} \lor \neg d_{p,a,q'} \tag{4.1}$$

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigvee_{q \in Q} d_{p,a,q} \tag{4.2}$$

Formula 4.1 makes sure that the variables $d_{p,a,q}$ in fact encode a well-defined function (i.e., for every state $p \in Q$ and input $a \in \Sigma$, there exists at most one $q \in Q$ such that $d_{p,a,q}$ is set to *true*). Formula 4.2, on the other hand, asserts that the transition function is total. The latter formula is not needed in general and might be skipped if the following definition is adjusted accordingly.

Let $\varphi_n^{\mathrm{DFA}}(\overline{d},\overline{f})$ be the conjunction of Formulas 4.1 and 4.2 where $\overline{d}$ is the list of all variables $d_{p,a,q}$ and $\overline{f}$ the list of all variables f_q for $p,q \in Q$ and $a \in \Sigma$.[5] Given a model $\mathfrak{M}$ of $\varphi_n^{\mathrm{DFA}}(\overline{d},\overline{f})$, deriving the corresponding DFA $\mathcal{A}_{\mathfrak{M}}$ is straightforward.

Definition 4.4. Let $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}}(\overline{d},\overline{f})$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q,\Sigma,q_0,\delta,F)$ is defined by

- $Q = \{q_0,\dots,q_{n-1}\}$;
- $\delta(p,a) = q$ for the unique q such that $d_{p,a,q}^{\mathfrak{M}} = \mathit{true}$; and
- $F = \{q \in Q \mid f_q^{\mathfrak{M}} = \mathit{true}\}$.

Note that Definition 4.4 remains sound if $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \psi$ where ψ can be an arbitrary formula in Propositional Boolean Logic. We exploit this fact and introduce three auxiliary formulas $\varphi_n^{\mathcal{A}^I}$, $\varphi_n^{\mathcal{A}^B}$, and $\varphi_n^{\mathcal{T}}$ in order to enforce that $\mathcal{A}_{\mathfrak{M}}$ is not just an arbitrary DFA but an IDFA. These formulas express the following:

- If $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{A}^I}$, then $L(\mathcal{A}^I) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.
- If $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{A}^B}$, then $L(\mathcal{A}^B) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.
- If $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{T}}$, then $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.

The idea behind the formulas $\varphi_n^{\mathcal{A}^I}$, $\varphi_n^{\mathcal{A}^B}$, and $\varphi_n^{\mathcal{T}}$ is to impose restrictions on the variables $d_{p,a,q}$ and f_q, which in turn determine the DFA $\mathcal{A}_{\mathfrak{M}}$. Keeping this in mind, it is easier to explain the formulas by directly referring to the automaton $\mathcal{A}_{\mathfrak{M}}$ rather than to describe their influence on the variables $d_{p,a,q}$ and f_q. Note, however, that we thereby implicitly assume that the corresponding formula is satisfiable and that $\mathfrak{M}$ is a model.

The idea of the formula $\varphi_n^{\mathcal{A}^I}$ is to keep track of the parallel behavior of the DFAs $\mathcal{A}^I$ and $\mathcal{A}_{\mathfrak{M}}$. To this end, we introduce new auxiliary variables $x_{q,q'}$ where $q \in Q$ and $q' \in Q_I$. Intuitively, we want to establish for all models $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \varphi_n^{\mathcal{A}^I}$ that if some input $u \in \Sigma^*$ induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} q$ and $\mathcal{A}_I : q_0^I \xrightarrow{u} q'$, then $x_{q,q'}$ has to be set to *true*. We achieve this using the following two formulas:

$$x_{q_0,q_0^I} \tag{4.3}$$

$$\bigwedge_{p,q \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{(p',a,q') \in \Delta_I} \left(x_{p,p'} \wedge d_{p,a,q}\right) \rightarrow x_{q,q'} \tag{4.4}$$

Formula 4.3 requires the variable x_{q_0,q_0^I} to be set to *true* because ε induces the runs $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}^I : q_0^I \xrightarrow{\varepsilon} q_0^I$. In addition, Formula 4.4 describes how to propagate the

[5] Recall that we use the short notation $\varphi(\overline{x})$ to indicate that the formula φ ranges over the (list of) variables $\overline{x} = (x_1,\dots,x_n)$.

values of the variables $x_{q,q'}$ step-by-step: if there exists a word u such that $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{u} p$ and $\mathcal{A}^I\colon q_0^I \xrightarrow{u} p'$ (i.e., the variable $x_{p,p'}$ is set to *true*), and there are transitions $\delta(p,a) = q$ in $\mathcal{A}_{\mathfrak{M}}$ and $(p', a, q') \in \Delta_I$, then $x_{q,q'}$ has to be set to *true*, too.

By using the variables $x_{q,q'}$, we can now translate the inclusion $L(\mathcal{A}^I) \subseteq L(\mathcal{A}_{\mathfrak{M}})$ as shown below by Formula 4.5. This formula expresses that whenever a word leads to an accepting state in $\mathcal{A}^I$, then it also has to lead to an accepting state in $\mathcal{A}_{\mathfrak{M}}$.

$$\bigwedge_{q \in Q} \bigwedge_{q' \in F_I} x_{q,q'} \rightarrow f_q \tag{4.5}$$

Let $\varphi_n^{\mathcal{A}^I}(\overline{d}, \overline{f}, \overline{x})$ be the conjunction of Formulas 4.3 to 4.5 where $\overline{d}$, $\overline{f}$ are as above and $\overline{x}$ is the list of new variables $x_{q,q'}$ with $q \in Q$ and $q' \in Q_I$. Then, we obtain the following result.

Lemma 4.2. *If $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}^I}(\overline{d}, \overline{f}, \overline{x})$, then $L(\mathcal{A}^I) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.*

Proof of Lemma 4.2. The proof follows the same line of arguments that we used in our intuitive description above. Let $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{A}^I}(\overline{d}, \overline{f}, \overline{x})$.

First, we show by induction that if there exists a word $u \in \Sigma^*$ such that $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{u} q$ and $\mathcal{A}^I\colon q_0^I \xrightarrow{u} q'$, then $x_{q,q'}$ is set to *true*.

Base case Let $u = \varepsilon$. The claim holds since $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{\varepsilon} q_0$ and $\mathcal{A}^I\colon q_0^I \xrightarrow{\varepsilon} q_0^I$ holds by definition and Formula 4.3 forces x_{q_0,q_0^I} to be set to *true*.

Induction step Let $u = va$ and assume $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{v} p \xrightarrow{a} q$ and $\mathcal{A}^I\colon q_0^I \xrightarrow{v} p' \xrightarrow{a} q'$. This particularly means that there exist transitions $\delta(p,a) = q$ in $\mathcal{A}_{\mathfrak{M}}$ (i.e., $d_{p,a,q}^{\mathfrak{M}} = \mathit{true}$) and $(p', a, q') \in \Delta_I$. Additionally, applying the induction hypothesis yields that $x_{p,p'}$ is set to *true*. Thus, Formula 4.4 forces $x_{q,q'}$ to be set to *true*, too.

Finally, let $u \in L(\mathcal{A}^I)$. Then, there exist $q \in Q$ and $q' \in F_I$ such that $\mathcal{A}_{\mathfrak{M}}\colon q_0 \xrightarrow{u} q$ and $\mathcal{A}^I\colon q_0^I \xrightarrow{u} q'$. Hence, $x_{q,q'}^{\mathfrak{M}} = \mathit{true}$. In this case, Formula 4.5 asserts $f_q^{\mathfrak{M}} = \mathit{true}$ and, hence, $q \in F$. Thus, $u \in L(\mathcal{A}_{\mathfrak{M}})$. □

We define the formula $\varphi_n^{\mathcal{A}^B}$ analogously to $\varphi_n^{\mathcal{A}^I}$. We introduce new auxiliary variables $y_{q,q'}$ (which have a similar semantics as the variables $x_{q,q'}$) where $q \in Q, q' \in Q_B$ and modify Formulas 4.3 and 4.4 accordingly. We also need to change Formula 4.5 slightly such that it now expresses that whenever a word is accepted by $\mathcal{A}^B$, it is rejected by $\mathcal{A}_{\mathfrak{M}}$. The resulting formulas are listed below:

$$y_{q_0,q_0^B} \tag{4.6}$$

$$\bigwedge_{p,q\in Q}\ \bigwedge_{a\in\Sigma}\ \bigwedge_{(p',a,q')\in\Delta_B} \left(y_{p,p'} \wedge d_{p,a,q}\right) \rightarrow y_{q,q'} \tag{4.7}$$

$$\bigwedge_{q\in Q}\ \bigwedge_{q'\in F_B} y_{q,q'} \rightarrow \neg f_q \tag{4.8}$$

Let $\varphi_n^{\mathcal{A}^B}(\overline{d},\overline{f},\overline{y})$ be the conjunction of Formulas 4.6 to 4.8 where $\overline{d}$, $\overline{f}$ are as above and $\overline{y}$ is the list of new variables $y_{q,q'}$ with $q \in Q$ and $q' \in Q_B$. Analogously to Lemma 4.2, we obtain the following result.

Lemma 4.3. *If* $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d},\overline{f}) \wedge \varphi_n^{\mathcal{A}^B}(\overline{d},\overline{f},\overline{y})$, *then* $L(\mathcal{A}^B) \cap L(\mathcal{A}_\mathfrak{M}) = \emptyset$.

Finally, the formula $\varphi_n^{\mathcal{T}}$ needs to enforce that $L(\mathcal{A}_\mathfrak{M})$ is inductive with respect to $R(\mathcal{T})$. To this end, we consider the parallel runs of $\mathcal{A}_\mathfrak{M}$ and $\mathcal{T}$ analogously to the formulas $\varphi_n^{\mathcal{A}^I}$ and $\varphi_n^{\mathcal{A}^B}$ above. This time, however, the situation is more involved since $\mathcal{T}$ works on pairs of words.

More precisely, we need to establish that if the pair $(u,v) \in \Sigma^* \times \Sigma^*$ is accepted by $\mathcal{T}$ and $u \in L(\mathcal{A}_\mathfrak{M})$, then $v \in L(\mathcal{A}_\mathfrak{M})$ holds as well. In other words, if $\mathcal{T}$ reaches a final state after reading (u,v) and $\mathcal{A}_\mathfrak{M}$ reaches a final state after reading u, then $\mathcal{A}_\mathfrak{M}$ must also reach a final state after reading v. We express this condition by means of auxiliary variables $z_{q,q',q''}$ where $q,q'' \in Q$ and $q' \in Q_T$. Their meaning is that if $\mathcal{T}: q_0^T \xrightarrow{(u,v)} q'$, $\mathcal{A}_\mathfrak{M}: q_0 \xrightarrow{u} q$, and $\mathcal{A}_\mathfrak{M}: q_0 \xrightarrow{v} q''$, then $z_{q,q',q''}$ is set to *true*.

The constraints on the variables $z_{q,q',q''}$ are then as follows:

$$z_{q_0,q_0^T,q_0} \tag{4.9}$$

$$\bigwedge_{p,p'',q,q''\in Q}\ \bigwedge_{a,b\in\Sigma}\ \bigwedge_{(p',(a,b),q')\in\Delta_T} \left(z_{p,p',p''} \wedge d_{p,a,q} \wedge d_{p'',b,q''}\right) \rightarrow z_{q,q',q''} \tag{4.10}$$

$$\bigwedge_{p,p'',q\in Q}\ \bigwedge_{a\in\Sigma}\ \bigwedge_{(p',(a,\varepsilon),q')\in\Delta_T} \left(z_{p,p',p''} \wedge d_{p,a,q}\right) \rightarrow z_{q,q',p''} \tag{4.11}$$

$$\bigwedge_{p,p'',q''\in Q}\ \bigwedge_{b\in\Sigma}\ \bigwedge_{(p',(\varepsilon,b),q')\in\Delta_T} \left(z_{p,p',p''} \wedge d_{p'',b,q''}\right) \rightarrow z_{p,q',q''} \tag{4.12}$$

$$\bigwedge_{q,q''\in Q}\ \bigwedge_{q'\in F_T} \left(z_{q,q',q''} \wedge f_q\right) \rightarrow f_{q''} \tag{4.13}$$

Formula 4.9 ensures that z_{q_0,q_0^B,q_0} is always set to *true*, Formulas 4.10 to 4.12 describe how the parallel runs of $\mathcal{A}_\mathfrak{M}$ and $\mathcal{T}$ evolve step-by-step, and Formula 4.13 makes sure that whenever (u,v) is accepted by $\mathcal{T}$ and $u \in L(\mathcal{A}_\mathfrak{M})$, then also $v \in L(\mathcal{A}_\mathfrak{M})$.

Let $\varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{z})$ be the conjunction of Formulas 4.9 to 4.13 where $\overline{d}$, $\overline{f}$ are as above and $\overline{z}$ is the list of new variables $z_{q,q',q''}$ with $q,q'' \in Q$ and $q' \in Q_T$. Then, we obtain the following result, which one can prove in the same way as Lemma 4.2.

Lemma 4.4. *If* $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d}, \overline{f}) \land \varphi_n^{\mathcal{T}}(\overline{d}, \overline{f}, \overline{z})$, *then* $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.

We now combine all subformulas and obtain the following result.

Lemma 4.5. *Let* $\mathcal{A}^I$, $\mathcal{A}^B$ *be two NFAs,* $\mathcal{T}$ *an asynchronous transducer,* $n \in \mathbb{N}_+$, *and*

$$\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}, \overline{z}) := \varphi_n^{DFA}(\overline{d}, \overline{f}) \land \varphi_n^{\mathcal{A}^I}(\overline{d}, \overline{f}, \overline{x}) \land \varphi_n^{\mathcal{A}^B}(\overline{d}, \overline{f}, \overline{y}) \land \varphi_n^{\mathcal{T}}(\overline{d}, \overline{f}, \overline{z}).$$

Then, the following statements hold:

1. *If* $\mathfrak{M} \models \varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}, \overline{z})$, *then* $\mathcal{A}_{\mathfrak{M}}$ *is an IDFA with n states; that is,* $\mathcal{A}_{\mathfrak{M}}$ *satisfies* $L(\mathcal{A}^I) \subseteq L(\mathcal{A}_{\mathfrak{M}})$, $L(\mathcal{A}^B) \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$, *and* $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$.
2. *If an IDFA* $\mathcal{A}$ *with n states satisfying* $L(\mathcal{A}^I) \subseteq L(\mathcal{A})$, $L(\mathcal{A}^B) \cap L(\mathcal{A}) = \emptyset$, *as well as* $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$ *exists, then* $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}, \overline{z})$ *is satisfiable.*

Proof of Lemma 4.5. The proof of Statement 1 follows from the properties of the formulas $\varphi_n^{\mathcal{A}^I}$, $\varphi_n^{\mathcal{A}^B}$, and $\varphi_n^{\mathcal{T}}$ (see Lemmas 4.2 to 4.4).

To prove the second statement, let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be an IDFA with n states that satisfies $L(\mathcal{A}^I) \subseteq L(\mathcal{A})$, $L(\mathcal{A}^B) \cap L(\mathcal{A}) = \emptyset$, and $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$. We can derive a model $\mathfrak{M}$ of $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}, \overline{z})$ as follows: we set $d_{p,a,q}^{\mathfrak{M}} = \mathit{true}$ if and only if $\delta(p,a) = q$, and $f_q^{\mathfrak{M}} = \mathit{true}$ if and only if $q \in F$; moreover, we derive an interpretation of the variables $x_{q,q'}$, $y_{q,q'}$, and $z_{q,q',q''}$ by looking at the states reached in the standard products of $\mathcal{A}$ and $\mathcal{A}^I$, $\mathcal{A}$ and $\mathcal{A}^B$, as well as $\mathcal{A}$ and $\mathcal{T}$, respectively. □

Finally, let us note that $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ can easily be turned into conjunctive normal form by applying De Morgan's law and the fact that $A \rightarrow B$ is logically equivalent to $\neg A \lor B$. Hence, we obtain the following remark about the number of variables and clauses of $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$.

Remark 4.2. In conjunctive normal form, the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ ranges over

$$\mathcal{O}\Big(n^2|\Sigma| + n(|Q_I| + |Q_B| + n|Q_T|)\Big)$$

variables and comprises

$$\mathcal{O}\Big(n^3|\Sigma| + n^2(|\Delta^I| + |\Delta^B| + n^2|\Delta^T|) + n(|F^I| + |F^B| + n|F^T|)\Big)$$

clauses.

4.2.2 Computing IDFAs Using the Logic of Uninterpreted Functions over the Natural Numbers

Next, we implement the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ in the logic of uninterpreted functions over the natural numbers. To this end, let us assume that all automata have a special form:

the set of states is $Q = [n]$, the initial state is 0, and the input alphabet is $\Sigma = [m]$. We can easily achieve this form by renaming the states of the automaton and symbols of the alphabet.

The formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ is based on the same ideas as in Section 4.2.1 but uses two uninterpreted functions $d\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ and $f\colon \mathbb{N}\to\mathbb{N}$ to encode the prospective DFA. The function d corresponds to the transition function and maps a pair of source-state and input-symbol to a destination-state; to ensure that d encodes a well-defined transition function, we impose constraints ensuring that $d(i,a) < n$ is satisfied for every $i \in Q$ and $a \in \Sigma$. The function f, on the other hand, encodes the set of final states; that is, we interpret the function values of f as Boolean values where 0 represents *false* and all other values represent *true* (i.e., $f(i) \neq 0$ indicates that i is a final state, whereas $f(i) = 0$ indicates that i is not a final state).

Given a model $\mathfrak{M} \models \varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$, it is straightforward to derive the DFA $\mathcal{A}_{\mathfrak{M}}$.

Definition 4.5. Let $\mathfrak{M} \models \varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$. The DFA $\mathcal{A}_{\mathfrak{M}} = (Q,\Sigma,q_0,\delta,F)$ is defined by

- $Q = [n]$;
- $q_0 = 0$;
- $\delta(i,a) = d^{\mathfrak{M}}(i,a)$ where $i \in Q$ and $a \in \Sigma$; and
- $F = \{i \in Q \mid f^{\mathfrak{M}}(i) \neq 0\}$.

To construct the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ (which asserts that $\mathcal{A}_{\mathfrak{M}}$ is an IDFA), we use three auxiliary uninterpreted functions $x\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$, $y\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$, and $z\colon \mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{N}$, which have the same meaning as the equally named Boolean variables of Section 4.2.1. Again, we interpret the function values of x, y, and z as *false* if they are 0 and otherwise as *true*.

Formulating the constraints of the previous section by means of the functions d, f, x, y, z is now immediate. At first, let $\varphi_n^{\mathrm{DFA}}(d)$ be Formula 4.14:

$$\bigwedge_{i\in Q}\bigwedge_{a\in\Sigma} d(i,a) < n \tag{4.14}$$

Next, let $\varphi_n^{\mathcal{A}^I}(d,f,x)$ be the conjunction of Formulas 4.15 to 4.17:

$$x(0,0) = 1 \tag{4.15}$$

$$\bigwedge_{i\in Q}\bigwedge_{a\in\Sigma}\bigwedge_{(i',a,j')\in\Delta_I} x(i,i') \neq 0 \rightarrow x(d(i,a),j') \neq 0 \tag{4.16}$$

$$\bigwedge_{i\in Q}\bigwedge_{i'\in F_I} x(i,i') \neq 0 \rightarrow f(i) \neq 0 \tag{4.17}$$

Moreover, let $\varphi_n^{\mathcal{A}^B}(d, f, y)$ the conjunction of Formulas 4.18 to 4.20:

$$y(0,0) = 1 \tag{4.18}$$

$$\bigwedge_{i \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{(i',a,j') \in \Delta_B} y(i,i') \neq 0 \rightarrow y(d(i,a), j') \neq 0 \tag{4.19}$$

$$\bigwedge_{i \in Q} \bigwedge_{i' \in F_B} y(i,i') \neq 0 \rightarrow f(i) = 0 \tag{4.20}$$

Finally, let $\varphi_n^{\mathcal{T}}(d, f, z)$ be the conjunction of Formulas 4.21 to 4.25:

$$z(0,0,0) = 1 \tag{4.21}$$

$$\bigwedge_{i,i'' \in Q} \bigwedge_{a,b \in \Sigma} \bigwedge_{(i',(a,b),j') \in \Delta_T} z(i,i',i'') \neq 0 \rightarrow z(d(i,a), j', d(i'',b)) \neq 0 \tag{4.22}$$

$$\bigwedge_{i,i'' \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{(i',(a,\varepsilon),j') \in \Delta_T} z(i,i',i'') \neq 0 \rightarrow z(d(i,a), j', i'') \neq 0 \tag{4.23}$$

$$\bigwedge_{i,i'' \in Q} \bigwedge_{b \in \Sigma} \bigwedge_{(i',(\varepsilon,b),j') \in \Delta_T} z(i,i',i'') \neq 0 \rightarrow z(i, j', d(i'',b)) \neq 0 \tag{4.24}$$

$$\bigwedge_{i,i'' \in Q} \bigwedge_{i' \in F_T} (z(i,i',i'') \neq 0 \wedge f(i) \neq 0) \rightarrow f(i'') \neq 0 \tag{4.25}$$

Then,

$$\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(d,f,x,y,z) := \varphi_n^{\mathrm{DFA}}(d) \wedge \varphi_n^{\mathcal{A}^I}(d,f,x) \wedge \varphi_n^{\mathcal{A}^B}(d,f,y) \wedge \varphi_n^{\mathcal{T}}(d,f,z)$$

is the desired formula, and we obtain the following lemma. Its proof is analogous to the proof of Lemma 4.5 (on Page 99) and, therefore, skipped.

Lemma 4.6. *Let $\mathcal{A}^I$, $\mathcal{A}^B$ be two NFAs, $\mathcal{T}$ an asynchronous transducer, $n \in \mathbb{N}_+$, and*

$$\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(d,f,x,y,z) := \varphi_n^{DFA}(d) \wedge \varphi_n^{\mathcal{A}^I}(d,f,x) \wedge \varphi_n^{\mathcal{A}^B}(d,f,y) \wedge \varphi_n^{\mathcal{T}}(d,f,z).$$

Then, the following statements hold:

1. *If $\mathfrak{M} \models \varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(d,f,x,y,z)$, then $\mathcal{A}_{\mathfrak{M}}$ is an IDFA with n states.*
2. *If an IDFA $\mathcal{A}$ with n states satisfying $L(\mathcal{A}^I) \subseteq L(\mathcal{A})$, $L(\mathcal{A}^B) \cap L(\mathcal{A}) = \emptyset$, as well as $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$ exists, then $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(d,f,x,y,z)$ is satisfiable.*

Finally, let us briefly comment on the size of $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}(d,f,x,y,z)$ when implemented in the logic of uninterpreted functions over the natural numbers.

Remark 4.3. The formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ ranges over five uninterpreted functions and comprises

$$\mathcal{O}\Big(n|\Sigma| + n(|\Delta_I| + |\Delta_B| + n|\Delta_T|) + n(|F_I| + |F_B| + n|F_T|)\Big)$$

constraints.

4.3 Semi-black-box Algorithms

Building on top of our black-box algorithms, we now develop two learning-based algorithms for Regular Model Checking that learn an invariant rather than compute one. The underlying idea is to abstract from the precise sets I and B (while we still require access to the program's transducer). This has the advantage that the complexity of the resulting algorithms does not immediately depend on the sets I and B but rather on the size of the learned DFA (and the transducer), which is in particular useful if the automata accepting I and B are large. Moreover, one can even apply the algorithms of this section in situations in which I and B are not regular, provided the teacher is still able to answer queries.

The learning takes place between an active learner, which has access to the program's transducer and wants to learn an invariant (in the form of an IDFA), and a teacher, who has access to the sets I, B and pretends to know an invariant. As in Angluin's original setting, the learner may pose membership and equivalence queries: a membership query corresponds to the question of whether a configuration belongs to an invariant, whereas an equivalence query asks whether a conjecture accepts a valid invariant. The problem is, however, that the teacher does not know an invariant and cannot answer certain queries. So as to still be able to apply active learning, we adapt Angluin's original setting to one in which answering queries is possible. In this new setting, a teacher answers membership queries with respect to I and B; moreover, if the configuration in question does not belong to either set, the teacher returns "don't know" because he cannot know in advance whether such a configuration should be included or excluded. On equivalence queries, on the other hand, a teacher only accepts conjectures that are inductive with respect to the transition relation. Then, answering an equivalence query amounts to checking whether the conjecture classifies the sets I and B correctly, which we assume the teacher can do. A formal definition of such a teacher, which we call *incomplete teacher*, is as follows.

Definition 4.6 (Incomplete teacher for Regular Model Checking)**.** Let $\mathcal{P} = (I, T)$ be a program over an alphabet Σ and $B \subseteq \Sigma^*$ with $I \cap B = \emptyset$ a set of bad configurations. An *incomplete teacher* for $\mathcal{P}$ and B answers queries as follows.

Membership query On an membership query with a word $u \in \Sigma^*$, the teacher replies "yes" if $u \in I$, "no" if $u \in B$, and "?" in any other case.

Equivalence query On an equivalence query, the teacher only accepts DFAs that are inductive with respect to the transitions T. When confronted with a conjecture $\mathcal{A}$, the teacher checks whether $I \subseteq L(\mathcal{A})$ and $B \cap L(\mathcal{A}) = \emptyset$ holds. If both conditions are satisfied (and since an incomplete teacher requires all conjectures to be inductive), $\mathcal{A}$ is an IDFA, and the teacher returns "yes". If this is not the case, the teacher returns a counterexample $u \in I \setminus L(\mathcal{A})$ or $u \in B \cap L(\mathcal{A})$.

A learner who learns from an incomplete teacher requires the following as input:

- An incomplete teacher for a program $\mathcal{P} = (I, T)$ and a set B
- An asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$

Since such a learner still requires the transducer to be given explicitly, it falls in between white-box and black-box techniques. By abuse of terminology, we refer to the present setting as a *semi-black-box setting* and to algorithms working in this setting as *semi-black-box algorithms*.

The fact that we now work in a semi-black-box setting entails that we need to provide both an incomplete teacher and a corresponding learner. We begin in Section 4.3.1 by demonstrating how to built an incomplete teacher who has access to two NFAs accepting the sets I and B. Note, however, that the present setting does not require I and B to be given as automata; the sets might be given be in any form (implicitly or explicitly), provided one can build an incomplete teacher. Then, we present two learners capable of learning IDFAs from an incomplete teacher.

Our learners work by interweaving an active and a passive learning algorithm. On the one hand, the active learning algorithm is responsible for querying the teacher and organizing the learned information. The passive learning algorithm, on the other hand, is responsible for constructing inductive conjectures based on the information learned so far. More precisely, the interplay between both learning algorithms works as follows. The active learning algorithm poses membership queries until it gathered enough information. Then, it extracts a finite sample $\mathcal{S} = (S_+, S_-)$ of classified words and hands $\mathcal{S}$ over to the passive learning algorithm. Based on $\mathcal{S}$ and the transducer $\mathcal{T}$, the passive learning algorithm infers a smallest DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$. The active algorithm part then uses the resulting DFA for posing an equivalence query. The requirement that the passive learning algorithm produces *smallest* DFAs is needed to guarantee the termination of the overall procedure (provided an IDFA exists).

We present a suitable passive learning algorithm in Section 4.3.2. On top of that, we develop two learners capable of learning from an incomplete teacher. The first,

presented in Section 4.3.3, follows the successful *counterexample-guided abstraction refinement (CEGAR)* principle [CGJ+00]. The second, presented in Section 4.3.4, is based on Angluin's algorithm for learning regular languages.

Finally, let us mention that a similar scenario called *learning from inexperienced teachers* was investigated by Grinchtein, Leucker, and Piterman [GLP06] and subsequently by Leucker and Neider [LN12]. In the latter work, also the general ideas of a CEGAR-style and an Angluin-style learner were discussed. However, the inexperienced teacher setting is simpler and does not involve computing inductive automata.

4.3.1 An Incomplete Teacher for Regular Model Checking

We now describe how to build an incomplete teacher for Regular Model Checking who has direct access to an NFA $\mathcal{A}^I$ with $L(\mathcal{A}^I) = I$ and an NFA $\mathcal{A}^B$ with $L(\mathcal{A}^B) = B$.

The teacher works as follows.

Membership query On a membership query with $u \in \Sigma^*$, the teacher returns "yes" if $u \in L(\mathcal{A}^I)$, "no" if $u \in L(\mathcal{A}^B)$, and "?" in any other case.

Equivalence query On an equivalence query with a DFA $\mathcal{A}$, the teacher checks whether $L(\mathcal{A}^I) \subseteq L(\mathcal{A})$ and $L(\mathcal{A}^B) \cap L(\mathcal{A}) = \emptyset$ holds and returns "yes" if so. If this is not the case, he either returns a counterexample $u \in L(\mathcal{A}^I) \setminus L(\mathcal{A})$, or a counterexample $u \in L(\mathcal{A}^B) \cap L(\mathcal{A})$. This check is in fact enough to ensure that $\mathcal{A}$ is an IDFA since we assume that every conjecture is inductive.

Note that all these checks are efficiently decidable for regular languages.

4.3.2 Passive Learning of Inductive DFAs from Samples and Transducers

In this section, we are faced with the following modified passive learning task: given a sample $\mathcal{S} = (S_+, S_-)$ consisting of two finite stets $S_+, S_- \subseteq \Sigma^*$ and an asynchronous transducer $\mathcal{T} = (Q_\mathcal{T}, \Sigma, \Sigma, q_0^\mathcal{T}, \Delta_\mathcal{T}, F_\mathcal{T})$, compute a smallest DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$.

To accomplish this task, we combine algorithms for passive learning from Section 3.1 with the solver-based algorithm of Section 4.2.1. More precisely, we create and solve a sequence of logic formulas $\varphi_n^{\mathcal{S},\mathcal{T}}$ for $n = 1, 2, \ldots$ that postulate the existence of a DFA with n states that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$. Moreover, the formulas are designed such that a model of $\varphi_n^{\mathcal{S},\mathcal{T}}$ provides the information necessary to construct a DFA with the desired properties. We increase the value of n until $\varphi_n^{\mathcal{S},\mathcal{T}}$ becomes satisfiable. The whole procedure is shown in Algorithm 4.2.

Analogous to Section 4.2.1, we implement the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ in Propositional Boolean Logic and the logic of uninterpreted functions over the natural numbers. Before we do so, however, let us claim the correctness of Algorithm 4.2.

Algorithm 4.2: Passive learning algorithm for learning inductive DFAs.

Input: A sample $\mathcal{S} = (S_+, S_-)$ and an asynchronous transducer $\mathcal{T}$ over a common alphabet Σ.

1 $n \leftarrow 0$.
2 **repeat**
3 $\quad n \leftarrow n + 1$.
4 $\quad$ Construct and solve $\varphi_n^{\mathcal{S},\mathcal{T}}$.
5 **until** $\varphi_n^{\mathcal{S},\mathcal{T}}$ *is satisfiable with model* $\mathfrak{M}$.
6 **return** the DFA $\mathcal{A}_{\mathfrak{M}}$ constructed from $\mathfrak{M}$.

Lemma 4.7. *Let $\mathcal{S}$ be a sample and $\mathcal{T}$ an asynchronous transducer, both over the common alphabet Σ. If a DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ exists, say with k states, then Algorithm 4.2 terminates after at most k iterations and returns a smallest DFA with these properties.*

The proof of Lemma 4.7 follows from the properties of $\varphi_n^{\mathcal{S},\mathcal{T}}$ (see Lemma 4.9 on Page 106, respectively Lemma 4.10 on Page 107). Moreover, since we increase n by one in every iteration, Algorithm 4.2 in fact returns a smallest DFA with the desired properties after k iterations provided that such a DFA exists.

Implementing $\varphi_n^{\mathcal{S},\mathcal{T}}$ in Propositional Boolean Logic

We reuse the formulas $\varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f})$ and $\varphi_n^{\mathcal{T}}(\overline{d}, \overline{f}, \overline{z})$ from Section 4.2.1 in order to implement the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$ in Propositional Boolean Logic. Let us remind the reader that the formula $\varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f})$ ensures that a model $\mathfrak{M}$ encodes a deterministic automaton, and $\varphi_n^{\mathcal{T}}(\overline{d}, \overline{f}, \overline{z})$ enforces the prospective DFA to be inductive with respect to the transducer $\mathcal{T}$. Thereby, $\overline{d}$, $\overline{f}$, and $\overline{z}$ are lists of variables with the following meaning:

- $\overline{d}$ is the list of variables $d_{p,a,q}$ where $p, q \in Q$ are states of the prospective DFA and $a \in \Sigma$. The variables $d_{p,a,q}$ encode the transitions of $\mathcal{A}_{\mathfrak{M}}$.
- $\overline{f}$ is the list of variables f_q where $q \in Q$, which encode the final states of $\mathcal{A}_{\mathfrak{M}}$.
- $\overline{z}$ is the list of variables $z_{q,q',q''}$ where $q, q'' \in Q$ and q' is a state of $\mathcal{T}$. These variables are used to track the parallel behavior of $\mathcal{A}_{\mathfrak{M}}$ and $\mathcal{T}$.

Moreover, given a model $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}} \wedge \psi$ where ψ is an arbitrary formula in Propositional Boolean logic, we construct the DFA $\mathcal{A}_{\mathfrak{M}}$ according to Definition 4.4 (on Page 96).

To express that the prospective DFA is consistent with the sample $\mathcal{S} = (S_+, S_-)$, we introduce a new formula $\varphi_n^{\mathcal{S}}$, which is a variation of Heule and Verwer's formula $\mu_n^{\mathcal{S}}$

from Section 3.1.4 (see Page 53). Formally, the formula $\varphi_n^{\mathcal{S}}(\overline{d},\overline{f},\overline{x})$ is the conjunction of Formulas 4.26 to 4.29 below where $\overline{x}$ is a list of variables $x_{u,q}$ with $u \in \mathrm{Pref}(S_+ \cup S_-)$ and $q \in Q$.[6]

$$x_{\varepsilon,q_0} \tag{4.26}$$

$$\bigwedge_{u \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{q,q' \in Q, q \neq q'} \neg x_{u,q} \vee \neg x_{u,q'} \tag{4.27}$$

$$\bigwedge_{ua \in \mathrm{Pref}(S_+ \cup S_-)} \bigwedge_{p,q \in Q} (x_{u,p} \wedge d_{p,a,q}) \rightarrow x_{ua,q} \tag{4.28}$$

$$\left(\bigwedge_{u \in S_+} \bigwedge_{q \in Q} x_{u,q} \rightarrow f_q \right) \wedge \left(\bigwedge_{u \in S_-} \bigwedge_{q \in Q} x_{u,q} \rightarrow \neg f_q \right) \tag{4.29}$$

In contrast to Heule and Verwer's original formula, we here need to explicitly fix the initial state of the prospective DFA to q_0 (cf. Formula 4.26) because the formula $\varphi_n^{\mathcal{T}}$ assumes q_0 to be the initial state.

The following lemma states that $\varphi_n^{\mathcal{S}}$ has indeed the desired properties.

Lemma 4.8. *If $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d},\overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d},\overline{f},\overline{x})$, then $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$ and $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$.*

Proof. The proof is similar to the correctness proof of Heule and Verwer's method (cf. the proof of Theorem 3.5 on Page 54 and following). First, a standard induction over the length of words $u \in \mathrm{Pref}(S_+ \cup S_-)$ using Formulas 4.26 to 4.28 shows that $\mathcal{A}_{\mathfrak{M}} \colon q_0 \xrightarrow{u} q$ implies $x_{u,q}^{\mathfrak{M}} = \mathit{true}$ and $x_{u,q'}^{\mathfrak{M}} = \mathit{false}$ for all $q' \in Q \setminus \{q\}$. Given this fact, we deduce using Formula 4.29 that $u \in L(\mathcal{A}_{\mathfrak{M}})$ holds for all $u \in S_+$ and $u \notin L(\mathcal{A}_{\mathfrak{M}})$ holds for all $u \in S_-$. □

Finally, we define $\varphi_n^{\mathcal{S},\mathcal{T}}$ to be the conjunction

$$\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{z}) \coloneqq \varphi_n^{\mathrm{DFA}}(\overline{d},\overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d},\overline{f},\overline{x}) \wedge \varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{z}),$$

which leads to the result shown next.

Lemma 4.9. *Let $\mathcal{S} = (S_+, S_-)$ be a sample and $\mathcal{T}$ an asynchronous transducer, both over the common alphabet Σ. In addition, let $n \in \mathbb{N}_+$ and*

$$\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{z}) \coloneqq \varphi_n^{DFA}(\overline{d},\overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d},\overline{f},\overline{x}) \wedge \varphi_n^{\mathcal{T}}(\overline{d},\overline{f},\overline{z}).$$

Then, the following statements hold:

[6] The meaning of a variable $x_{u,q}$ is that it is set to *true* if the prospective DFA reaches state q after reading the input u. We encourage the reader to consult Section 3.1.4 for a detailed explanation.

1. *If* $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{z})$*, then* $\mathcal{A}_{\mathfrak{M}}$ *is a DFA with* n *states that satisfies* $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$*,* $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$*, and* $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$*.*
2. *If a DFA* $\mathcal{A}$ *with* n *states exists that satisfies* $S_+ \subseteq L(\mathcal{A})$*,* $S_- \cap L(\mathcal{A}) = \emptyset$*, as well as* $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$*, then* $\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{z})$ *is satisfiable.*

Lemma 4.9 follows from Lemma 4.4 (on Page 99) and Lemma 4.8. Moreover, we observe the following regarding the size of the formula $\varphi_n^{\mathcal{S},\mathcal{T}}$.

Remark 4.4. In conjunctive normal form, the formula $\varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d},\overline{f},\overline{x},\overline{z})$ ranges over

$$\mathcal{O}\Big(n^2|\Sigma| + n^2|Q_{\mathcal{T}}| + n|\mathrm{Pref}(S_+ \cup S_-)|\Big)$$

variables and comprises

$$\mathcal{O}\Big(n^3|\Sigma| + n^2(n^2|\Delta_{\mathcal{T}}| + |F_{\mathcal{T}}|) + n^2|\mathrm{Pref}(S_+ \cup S_-)|\Big)$$

clauses.

Implementing $\varphi_n^{\mathcal{S},\mathcal{T}}$ in the Logic of Uninterpreted Functions over the Natural Numbers

To implement $\varphi_n^{\mathcal{S},\mathcal{T}}$ in the logic of uninterpreted functions over the natural numbers, we combine the formulas φ_n^{DFA} and $\varphi_n^{\mathcal{T}}$ from Section 4.2.2 and formula $\nu_n^{\mathcal{S}}$ from Section 3.1.5 (on Page 55 and following). More precisely, $\varphi_n^{\mathcal{S},\mathcal{T}}$ is the conjunction

$$\varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z) \coloneqq \varphi_n^{\mathrm{DFA}}(d,f) \land \nu_n^{\mathcal{S}}(d,f,x) \land \varphi_n^{\mathcal{T}}(d,f,z)$$

where $d\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, $f\colon \mathbb{N} \to \mathbb{N}$, $x\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, and $z\colon \mathbb{N}\times\mathbb{N}\times\mathbb{N} \to \mathbb{N}$ are uninterpreted functions that have the same meaning as the corresponding Boolean variables of the previous section. Moreover, given a model $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z)$, we construct the DFA $\mathcal{A}_{\mathfrak{M}}$ analogous to Definition 4.5 (on Page 100).

The following lemma states that $\varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z)$ has in fact the desired properties.

Lemma 4.10. *Let* $\mathcal{S} = (S_+, S_-)$ *be a sample and* $\mathcal{T}$ *an asynchronous transducer, both over the common alphabet* Σ*. In addition, let* $n \in \mathbb{N}_+$ *and*

$$\varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z) \coloneqq \varphi_n^{DFA}(d,f) \land \nu_n^{\mathcal{S}}(d,f,x) \land \varphi_n^{\mathcal{T}}(d,f,z).$$

Then, the following statements hold:

1. *If* $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z)$*, then* $\mathcal{A}_{\mathfrak{M}}$ *is a DFA with* n *states that satisfies* $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$*,* $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$*, and* $R(\mathcal{T})(L(\mathcal{A}_{\mathfrak{M}})) \subseteq L(\mathcal{A}_{\mathfrak{M}})$*.*

2. *If a DFA $\mathcal{A}$ with n states exists that satisfies $S_+ \subseteq L(\mathcal{A})$, $S_- \cap L(\mathcal{A}) = \emptyset$, as well as $R(\mathcal{T})(L(\mathcal{A})) \subseteq L(\mathcal{A})$, then $\varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z)$ is satisfiable.*

Considering the properties of the individual subformulas of $\varphi_n^{\mathcal{S},\mathcal{T}}$, a proof of Lemma 4.10 is straightforward (cf. Theorem 3.6 on Page 56 and Lemma 4.6 on Page 101) and, therefore, skipped.

Finally, let us briefly comment on the size of of the formula $\varphi_n^{\mathcal{A}^I,\mathcal{A}^B,\mathcal{T}}$ when implemented in the logic of uninterpreted functions over the natural numbers.

Remark 4.5. The formula $\varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,z)$ ranges over four uninterpreted functions and comprises

$$\mathcal{O}\Big(n|\Sigma| + |\mathrm{Pref}(S_+ \cup S_-)| + n^2(|\Delta_T| + |F_T|)\Big)$$

constraints.

4.3.3 The CEGAR-style Learner

The *CEGAR-style learner*, sketched as Algorithm 4.3, maintains a sample $\mathcal{S} = (S_+, S_-)$ as a finite abstraction of the (potentially infinite) sets of initial and bad configurations. In every iteration, the learner computes a minimal DFA $\mathcal{A}$ that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ using Algorithm 4.2 (see Page 105). Subsequently, the learner submits $\mathcal{A}$ on an equivalence query. If the teacher replies "yes" to this query, the learning terminates. If the teacher returns a counterexample, say u, the learner classifies u by checking whether $u \in L(\mathcal{A})$ holds and refines its abstraction of the program (i.e., its sample $\mathcal{S}$): since a counterexample either satisfies $u \in I \setminus L(\mathcal{A})$ or $u \in B \cap L(\mathcal{A})$, the learner adds u to S_+ in the first case or u to S_- in the latter case. This excludes a spurious behavior of further conjectures on u. Then, the learner proceeds with the next iteration.

Algorithm 4.3 follows the CEGAR principle in the following sense. The DFA $\mathcal{A}$ produced from the sample in every iteration is an abstraction of the reachable configurations of the program. In the beginning, the sample contains only a few words and the CEGAR-style learner produces very coarse abstractions. An equivalence query with the abstraction reveals if an IDFA has been found. If this is not the case, counterexamples are used to refine the abstraction until an IDFA can be identified.

The following theorem states the correctness of Algorithm 4.3.

Theorem 4.11. *Let $\mathcal{P} = (I,T)$ be a program over the common alphabet Σ and $B \subseteq \Sigma^*$ a regular set of bad configurations with $B \cap I = \emptyset$. Moreover, let an incomplete teacher for $\mathcal{P}$ and B and an asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$ be given. If an IDFA with respect to $\mathcal{P}$ and B exists, Algorithm 4.3 terminates and returns a smallest IDFA.*

Algorithm 4.3: CEGAR-style learner for Regular Model Checking.

Input: An incomplete teacher for a program $\mathcal{P} = (I, T)$ and an asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$.

1 Initialize a sample $\mathcal{S} = (S_+, S_-)$ with $S_+ = \emptyset$ and $S_- = \emptyset$.
2 **repeat**
3 Compute a smallest DFA $\mathcal{A}$ consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ using Algorithm 4.2.
4 Perform an equivalence query with $\mathcal{A}$.
5 **if** *the teacher returns a counterexample* u **then**
6 **if** $u \in L(\mathcal{A})$ **then**
7 $S_- \leftarrow S_- \cup \{u\}$.
8 **else**
9 $S_+ \leftarrow S_+ \cup \{u\}$.
10 **end**
11 **end**
12 **until** *the teacher replies "yes" to the equivalence query with* $\mathcal{A}$.
13 **return** $\mathcal{A}$.

Proof of Theorem 4.11. Due to the way an incomplete teacher answers equivalence queries, we know that the result of Algorithm 4.3 is in fact an IDFA once the algorithm has terminated. Thus, it is enough to show that Algorithm 4.3 eventually terminates if an IDFA exists, and that its result is of minimal size.

To this end, we make three observations.

1. Algorithm 4.3 never conjectures the same DFA twice. This is due to the fact that the learner successively adds counterexamples to the sample and a conjecture is consistent with the sample of the iteration in which it is produced and all previous iterations. Thus, the conjecture $\mathcal{A}_i$ of iteration i and the conjecture $\mathcal{A}_j$ of iteration $j < i$ classify at least the counterexample added in iteration j differently.

2. The size of consecutive conjectures increases monotonically during the learning process. This can be seen as follows: Assume that the conjecture $\mathcal{A}_i$ has n_i states and the conjecture $\mathcal{A}_{i+1}$ has $n_{i+1} < n_i$ states. Since the sample of iteration $i + 1$ results from the one of iteration i by adding one word to S_+ or S_-, the conjecture $\mathcal{A}_{i+1}$ is necessarily consistent with the sample of iteration i but has fewer states than the conjecture $\mathcal{A}_i$. This contradicts the fact that the learner only produces minimal consistent (and inductive) DFAs.

3. Every IDFA, in particular every smallest IDFA, is consistent with the samples produced during the run of Algorithm 4.3 since the samples contain only coun-

terexamples whose classifications are known to the teacher; that is, adding counterexamples to the sample does not rule out any IDFA.

Now, suppose that an IDFA exists and assume that a smallest IDFA has k states. Since there exists no IDFA with less than k states and due to Observations 1 and 2, we know that Algorithm 4.3 eventually conjectures DFAs with at least k states. Towards a contradiction suppose that Algorithm 4.3 does not find an IDFA with k states and eventually conjectures a DFA with $k' > k$ states. Then, Observation 3 together with the fact that the learner always computes smallest consistent and inductive DFAs implies that there cannot be an IDFA with k states. This is a contradiction. Thus, Algorithm 4.3 eventually conjectures a smallest IDFA, which passes the equivalence query, and terminates. □

It is worth pointing out that Theorem 4.11 holds regardless of the specific counterexamples the teacher returns. Indeed, the mere fact that the CEGAR-style learner uses counterexamples to refine its abstraction of the program and that it always computes smallest consistent and inductive conjectures is enough to prove Theorem 4.11.

Finally, the proof of Theorem 4.11 suggests to start the passive learning algorithm not with $n = 1$ but with the size of the last conjecture. Since we know that there is no IDFA with less states, an initial value of $n > 1$ avoids unnecessary calls to the logic solver and improves the performance of Algorithm 4.3.

4.3.4 The Angluin-style Learner

Our Angluin-style learner is an adaptation of Angluin's original algorithm, which learns an IDFA from an incomplete teacher. Like the CEGAR-style learner, the Angluin-style learner uses a logic solver to find a smallest inductive DFA that is consistent with the information learned so far.

The Angluin-style learner stores its information in an *extended observation table* $O_? = (R, S, T_?)$ where $R \subseteq \Sigma^*$ is a prefix-closed set of representatives, $S \subseteq \Sigma^*$ is a set of separating words, and $T_? \colon (R \cup R \cdot \Sigma) \cdot S \to \{0, 1, ?\}$ is a mapping that stores the table entries; in contrast to Angluin's original algorithm, an extended observation table now also stores ?-entries. The learner fills the table using membership queries such that the value $T_?(u)$ corresponds to the teacher's answer to the membership query with the word u. This task is delegated to a subroutine *update*$(O_?)$, which queries the teacher for missing table entries and fills the table accordingly.

Like in Angluin's original algorithm, the representatives from R are candidates for identifying states, and the words from S are used to distinguish states. This time, however, the notion of equivalent representatives is no longer immediate due the existence of ?-entries in the table. Therefore, we follow Grinchtein, Leucker, and

Piterman's idea [GLP06] and call two words $u, v \in R \cup R \cdot \Sigma$ *similar*, denoted by $u \approx_{O_?} v$, if $T_?(uw) \neq ?$ and $T_?(vw) \neq ?$ implies $T_?(uw) = T_?(vw)$ for all $w \in S$; in other words, u and v cannot be distinguished by words from S. Note, however, that $\approx_{O_?}$ is not an equivalence relation, unlike in Angluin's algorithm.

The Angluin-style learner, shown as Algorithm 4.4 (on Page 112), works similar to Angluin's original algorithm, which makes the table closed and consistent in every iteration. Since the Angluin-style learner needs to handle ?-entries, we switch to weaker notions of closedness and consistency, which are based upon the relation $\approx_{O_?}$:

- An extended observation table $O_?$ is *weakly closed* if for all $u \in R$ and $a \in \Sigma$ there exists a $v \in R$ such that $ua \approx_{O_?} v$. If this is not satisfied, the Angluin-style learner adds ua to R and updates the table.

- An extended observation table $O_?$ is *weakly consistent* if for all $u, v \in R$ and $a \in \Sigma$, $u \approx_{O_?} v$ implies $ua \approx_{O_?} va$. If $O_?$ is not weakly consistent, there exist $u, v \in R$, $w \in S$, and $a \in \Sigma$ such that $T_?(uaw) \neq ?$, $T_?(vaw) \neq ?$, and $T_?(uaw) \neq T_?(vaw)$. In this situation, the Angluin-style learner adds aw to S and updates the table.

As in the case of Angluin's original algorithm, one can check both weak closedness and weak consistency of an extended observation table in time polynomial in the size of the table.

Once $O_?$ is weakly closed and weakly consistent, the Angluin-style learner turns the table into a sample $\mathcal{S} = (S_+, S_-)$ with

$$S_+ = \{u \mid T_?(u) = 1\} \text{ and } S_- = \{u \mid T_?(u) = 0\}.$$

Then, it applies Algorithm 4.2 (see Page 105) to derive a smallest DFA that is consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$, which it submits to an equivalence query. If the teacher replies "yes", the learning terminates. Otherwise, the Angluin-style learner adds the returned counterexample and all of its prefixes to R, updates the table, and proceeds with the next iteration.

The following theorem states the correctness of Algorithm 4.4.

Theorem 4.12. *Let $\mathcal{P} = (I, T)$ be a program over the common alphabet Σ and $B \subseteq \Sigma^*$ a regular set of bad configurations with $B \cap I = \emptyset$. Moreover, let an incomplete teacher for $\mathcal{P}$ and B and an asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$ be given. If an IDFA with respect to $\mathcal{P}$ and B exists, Algorithm 4.4 terminates and returns a smallest IDFA.*

The proof of this theorem is similar to correctness proof of the CEGAR-style learner (cf. Theorem 4.11 on Page 108), but slightly more involved since the Angluin-style learner stores its information in an extended observation table rather than in a sample.

Algorithm 4.4: Angluin-style learner for Regular Model Checking.

Input: An incomplete teacher for a program $\mathcal{P} = (I, T)$ and an asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$.

1 Initialize the extended observation table $O_? = (R, S, T_?)$ with $R = S = \{\varepsilon\}$, and *update*$(O_?)$.
2 **repeat**
3   **while** *$O_?$ is not weakly closed or not weakly consistent* **do**
4   **if** *$O_?$ is not weakly closed* **then**
5   Pick $u \in R$ and $a \in \Sigma$ such that there exists no $v \in R$ with $ua \approx_{O_?} v$.
6       $R \leftarrow R \cup \{ua\}$.
7   *update*$(O_?)$.
8   **end**
9   **if** *$O_?$ is not weakly consistent* **then**
10   Pick $u, v \in R$, $a \in \Sigma$, and $w \in S$ with $u \approx_{O_?} v$, $T_?(uaw) \neq ?$, $T_?(vaw) \neq ?$, and $T_?(uaw) \neq T_?(vaw)$.
11       $S \leftarrow S \cup \{aw\}$.
12   *update*$(O_?)$.
13   **end**
14   **end**
15   Let $\mathcal{S} = (\{u \mid T_?(u) = 1\}, \{u \mid T_?(u) = 0\})$.
16   Compute a smallest DFA $\mathcal{A}$ consistent with $\mathcal{S}$ and inductive with respect to $\mathcal{T}$ using Algorithm 4.2.
17   Perform an equivalence query with $\mathcal{A}$.
18   **if** *the teacher returns a counterexample u* **then**
19     $R \leftarrow R \cup \mathrm{Pref}(u)$.
20   *update*$(O_?)$.
21   **end**
22 **until** *the teacher replies "yes" to the equivalence query with $\mathcal{A}$.*
23 **return** $\mathcal{A}$.

Proof of Theorem 4.12. Like in the proof of Theorem 4.11, it is enough to show that Algorithm 4.3 terminates if an IDFA exists (the returned DFA is then guaranteed to be an IDFA) and that the resulting IDFA is of minimal size.

We begin with the remark that any conjecture $\mathcal{A}$ produced by the Angluin-style learner is *consistent with the extended observation table $O_?$*; that is, it satisfies

$$\{u \mid T_?(u) = 1\} \subseteq L(\mathcal{A}) \text{ and } \{u \mid T_?(u) = 0\} \cap L(\mathcal{A}) = \emptyset.$$

The reason is simple: the learner extracts a sample $\mathcal{S}$ consisting of exactly those two sets and produces a smallest DFA that is consistent with $\mathcal{S}$ (and inductive with respect to $\mathcal{T}$).

Next, we make three observations, which resemble those in the proof of Theorem 4.11.

1. Algorithm 4.4 never conjectures the same DFA twice. This is due to the fact that the learner successively adds counterexamples to the table (and perhaps performs further membership queries) and a conjecture is consistent with the table of the iteration in which it is produced and all previous iterations. Thus, the conjecture $\mathcal{A}_i$ of iteration i and the conjecture $\mathcal{A}_j$ of iteration $j < i$ classify at least the counterexample added in iteration j differently.

2. The size of consecutive conjectures increases monotonically: Suppose that the conjecture $\mathcal{A}_i$ of iteration i has n_i states and the conjecture $\mathcal{A}_{i+1}$ of iteration $i+1$ has $n_{i+1} < n_i$ states. Since the table of iteration $i+1$ results from the one of iteration i by adding at least one counterexample and its classification to the table, $\mathcal{A}_{i+1}$ is necessarily consistent with the table of iteration i but has fewer states than $\mathcal{A}_i$. This contradicts the fact that the learner only produces minimal consistent (and inductive) DFAs.

3. Every IDFA, in particular every smallest IDFA, is consistent with the tables created during the run of Algorithm 4.4 since the learner produces conjectures that are consistent with all non-?-entries.

Analogously to Theorem 4.11, we deduce that Algorithm 4.4 eventually conjectures a smallest IDFA, which passes the equivalence query, and terminates. □

4.4 Black-box Algorithms

Finally, we present two learning-based algorithms for Regular Model Checking that obtain their information solely from a teacher and, hence, completely work in a black-box fashion; that means in particular that a learner has no access to the program in question. In contrast to an incomplete teacher of the previous section, who assumes that the learner knows the transducer and produces inductive conjectures, a teacher now has to communicate all information about the program that are necessary to enable the learner to eventually learn an invariant (provided one exists). The problem is, however, that the teacher does not know an invariant.

As a solution, we have recently proposed the *ICE-learning framework* [GLMN13a, GLMN14], which is a passive learning setting (without minimality constraint). In order to interact with a teacher, we have also introduced an iterative version, called *iterative ICE-learning*, in which a learner and a teacher communicate via equivalence queries (which are also called *correctness queries* to emphasize that there does not exist a unique target concept to which a conjecture could be equivalent). On an

equivalence query, the learner proposes a conjecture, and the teacher checks whether this conjecture accepts an invariant. If this is not the case, the teacher either returns a (classical) counterexample witnessing that the conjecture classifies an initial or a bad configuration incorrectly, or an implication-counterexample witnessing that the conjecture is not inductive; recall that an implication-counterexample is a pair (c, c') of program configurations such that the program can move from configuration c to configuration c', c is accepted by the conjecture, and c' is not accepted. It seems that this is the maximal amount of information a teacher without actual knowledge of an invariant can provide on an equivalence query.

For the present setting, we combine the incomplete teacher of Section 4.3 (for answering membership queries) and iterative ICE-learning (for answering equivalence queries). A corresponding teacher, which we call *ICE-teacher*, works as follows.

Definition 4.7 (ICE-teacher for Regular Model Checking)**.** Let $\mathcal{P} = (I, T)$ be a program over the alphabet Σ and $B \subseteq \Sigma^*$ with $I \cap B = \emptyset$ a set of bad configurations. An *ICE-teacher* for $\mathcal{P}$ and B answers queries as follows.

Membership query On a membership query with a word $u \in \Sigma^*$, the teacher replies "yes" if $u \in I$, "no" if $u \in B$, and "don't know", denoted by "?", in any other case.

Equivalence query On an equivalence query with a DFA $\mathcal{A}$, the teacher performs the following tasks:

1. If $I \not\subseteq L(\mathcal{A})$, then the teacher returns a counterexample $u \in I \setminus L(\mathcal{A})$.
2. If $B \cap L(\mathcal{A}) \neq \emptyset$, then the teacher returns a counterexample $u \in B \cap L(\mathcal{A})$.
3. If $\mathcal{A}$ is not inductive, the teacher returns an implication-counterexample (u, u') with $(u, u') \in T$, $u \in L(\mathcal{A})$, and $u' \notin L(\mathcal{A})$.
4. If $\mathcal{A}$ passes the checks above, $\mathcal{A}$ is an IDFA, and the teacher returns "yes".

The order in which the teacher performs Tasks 1 to 3 is arbitrary.

Note that incomplete teachers and ICE-teachers (only) differ in the way they answer equivalence queries: an incomplete teacher expects an inductive DFA and returns classical counterexamples, whereas an ICE-teacher does not pose any restriction on a conjecture and returns either classical counterexamples or implication-counterexamples.

In Section 4.4.1, we demonstrate how to construct an ICE-teacher who is based on a representation of a program in terms of finite automata. Note, however, that the present setting does not require a program to be given in terms of automata. Instead, it can be given in any form provided one can build an ICE-teacher.

In Sections 4.4.3 and 4.4.4, we lift the CEGAR-style learner of Section 4.3.3, respectively the Angluin-style learner of Section 4.3.4, to the ICE-teacher setting. We

call the resulting learners *CEGAR-style ICE-learner* and *Angluin-style ICE-learner*. The underlying idea is to record implication-counterexamples returned on equivalence queries in a (finite) set $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ and to replace the passive learning algorithm, which is responsible for producing conjectures. The new passive learning algorithm takes a finite sample $\mathcal{S}$ as well as a finite set of implications $\mathcal{C}$ and produces a smallest DFA $\mathcal{A}$ that is consistent with $\mathcal{S}$ and *respects the implications in* $\mathcal{C}$; the latter means that $u \in L(\mathcal{A})$ implies $u' \in L(\mathcal{A})$ for all $(u, u') \in \mathcal{C}$. We present such a passive learning algorithm in Section 4.4.2. Our black-box algorithm for Regular Model Checking is then simply the bundle consisting of an ICE-teacher and one of these learners.

4.4.1 An ICE-teacher for Regular Model Checking

We now describe how to build an ICE-teacher for Regular Model Checking who has direct access to an automaton representation of the program $\mathcal{P} = (I, T)$ and the set B of bad configurations; that is, we assume that the teacher has access to an NFA $\mathcal{A}^I$ with $L(\mathcal{A}^I) = I$, an NFA $\mathcal{A}^B$ with $L(\mathcal{A}^B) = B$, and an asynchronous transducer $\mathcal{T}$ with $R(\mathcal{T}) = T$.

The ICE-teacher works as follows.

Membership query On a membership query with $u \in \Sigma^*$, the teacher returns “yes” if $u \in L(\mathcal{A}^I)$, “no” if $u \in L(\mathcal{A}^B)$, and “?” in any other case.

Equivalence query On an equivalence query with a DFA $\mathcal{A}$, the teacher performs the following tasks:

1. If $L(\mathcal{A}^I) \nsubseteq L(\mathcal{A})$, the teacher returns a counterexample $u \in L(\mathcal{A}^I) \setminus L(\mathcal{A})$.
2. If $L(\mathcal{A}^B) \cap L(\mathcal{A}) \neq \emptyset$, the teacher returns a counterexample $u \in L(\mathcal{A}^B) \cap L(\mathcal{A})$.
3. To check whether $\mathcal{A}$ is inductive, the teacher computes a DFA $\mathcal{A}'$ with $L(\mathcal{A}') = R(\mathcal{T})(L(\mathcal{A}))$ according to Lemma 2.1 (see Page 17). If $L(\mathcal{A}') \subseteq L(\mathcal{A})$ does not hold, then the teacher computes a DFA $\mathcal{A}''$ such that $L(\mathcal{A}'') = L(\mathcal{A}') \cap (\Sigma^* \setminus L(\mathcal{A}))$; that is, $\mathcal{A}''$ accepts all configurations $u' \notin L(\mathcal{A})$ for which a configuration $u \in L(\mathcal{A})$ with $(u, u') \in R(\mathcal{T})$ exists. Finally, the teacher chooses an arbitrary configuration $u' \in L(\mathcal{A}'')$ and constructs a corresponding configuration u such that the run of $\mathcal{T}$ on the pair (u, u') is accepting. Once the teacher identified such a pair, he returns it as an implication-counterexample.
4. If $\mathcal{A}$ passes the checks above, $\mathcal{A}$ is an IDFA, and the teacher returns “yes”.

Note that answering both membership and equivalence queries is possible using standard methods of automata theory.

4.4.2 Passive Learning of DFAs from Samples and Implications

In this section, we present an algorithm for the following modified passive learning task: given a sample $\mathcal{S} = (S_+, S_-)$ over an alphabet Σ and a finite set $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ of implications, compute a smallest DFA that is consistent with $\mathcal{S}$ and respects the implications in $\mathcal{C}$.

As before, we use a logic solver to solve this task. Our algorithm creates and solves once again a sequence of logic formulas $\varphi_n^{\mathcal{S},\mathcal{C}}$ for $n = 1, 2, \ldots$ that postulate the existence of a DFA with n states that is consistent with S and respects the implications in $\mathcal{C}$. We design $\varphi_n^{\mathcal{S},\mathcal{C}}$ such that a model $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{C}}$ provides the information necessary to construct a DFA with the desired properties. Algorithm 4.5 shows the resulting algorithm in pseudo code.

Algorithm 4.5: Passive learning algorithm for learning DFAs that respect implications.

Input: A sample $\mathcal{S} = (S_+, S_-)$ over Σ and a finite set $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ of implications.

```
1 n ← 0.
2 repeat
3 |  n ← n + 1.
4 |  Construct and solve φ_n^{S,C}.
5 until φ_n^{S,C} is satisfiable with model 𝔐.
6 return the DFA A_𝔐 constructed from 𝔐.
```

The lemma below states the correctness of Algorithm 4.5.

Lemma 4.13. *Let $\mathcal{S}$ be a sample over Σ and $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ a finite set of implications. If a DFA that is consistent with $\mathcal{S}$ and respects the implications in $\mathcal{T}$ exists, say with k states, then Algorithm 4.5 terminates after at most k iterations and returns a smallest DFA with these properties.*

Again, the proof of Lemma 4.13 is a straightforward application of the properties of $\varphi_n^{\mathcal{S},\mathcal{C}}$ (cf. Lemma 4.15 on Page 118, respectively Lemma 4.16 on Page 119). Moreover, since we increase n by one in every iteration, Algorithm 4.5 indeed finds a smallest DFA with the desired properties after k iterations provided that such a DFA exists.

Thus, it is left to implement the formula $\varphi_n^{\mathcal{S},\mathcal{C}}$. Again, we do so in Propositional Boolean Logic and in the logic of uninterpreted functions over the natural numbers.

Implementing $\varphi_n^{\mathcal{S},\mathcal{C}}$ in Propositional Boolean Logic

The first step towards implementing the formula $\varphi_n^{\mathcal{S},\mathcal{C}}$ in Propositional Boolean Logic is to construct a formula $\varphi_n^{\mathcal{C}}$ that expresses that the prospective DFA respects the

implications in $\mathcal{C}$. To this end, let

$$C = \{u \mid \exists u' \colon (u, u') \in \mathcal{C}\} \cup \{u' \mid \exists u \colon (u, u') \in \mathcal{C}\}$$

be the set of all words occurring as either antecedent or as consequent of an implication in $\mathcal{C}$.

The idea of $\varphi_n^{\mathcal{C}}$ is to encode the runs of the prospective DFA $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0, \delta, F)$ on the words in $\mathrm{Pref}(C)$ and to enforce that $u \in L(\mathcal{A}_{\mathfrak{M}})$ implies $u' \in L(\mathcal{A}_{\mathfrak{M}})$ for all $(u, u') \in \mathcal{C}$. To this end, we use auxiliary Boolean variables $y_{u,q}$ where $u \in \mathrm{Pref}(C)$ and $q \in Q$. The meaning of a variable $y_{u,q}$ is that it is set to *true* if $\mathcal{A}_{\mathfrak{M}}$ reaches state q after reading the word u. Analogously to formula φ_n^S in Propositional Boolean Logic, we achieve this using the following three formulas.

$$y_{\varepsilon,q_0} \tag{4.30}$$

$$\bigwedge_{u \in \mathrm{Pref}(C)} \bigwedge_{q,q' \in Q, q \neq q'} \neg y_{u,q} \lor \neg y_{u,q'} \tag{4.31}$$

$$\bigwedge_{ua \in \mathrm{Pref}(C)} \bigwedge_{p,q \in Q} (y_{u,p} \land d_{p,a,q}) \rightarrow y_{ua,q} \tag{4.32}$$

Recall that the variables $d_{p,a,q}$ where $p, q \in Q$ and $a \in \Sigma$ are used to encode the transitions of the prospective DFA.

To enforce that $\mathcal{A}_{\mathfrak{M}}$ respects the implications in $\mathcal{C}$, we need to express the following: for all implications $(u, u') \in \mathcal{C}$, whenever $\mathcal{A}_{\mathfrak{M}}$ reaches a final state, say p, after reading u, then $\mathcal{A}_{\mathfrak{M}}$ must also reach a final state, say q, after reading u'. This conditions is expressed by the formula below.

$$\bigwedge_{(u,u') \in \mathcal{C}} \bigwedge_{p \in Q} \left((y_{u,p} \land f_p) \rightarrow \bigwedge_{q \in Q} (y_{u',q} \rightarrow f_q) \right) \tag{4.33}$$

As before, the variables f_q where $q \in Q$ encode the final states of the prospective DFA.

Formula 4.33 is not in conjunctive normal form. However, applying the distributive law yields the following equivalent formula, which is in conjunctive normal form.

$$\bigwedge_{(u,u') \in \mathcal{C}} \bigwedge_{p \in Q} \bigwedge_{q \in Q} \left(\neg y_{u,p} \lor \neg f_p \lor \neg y_{u',q} \lor f_q \right) \tag{4.34}$$

Let $\varphi_n^{\mathcal{C}}(\overline{d}, \overline{f}, \overline{y})$ be the conjunction of Formulas 4.30 to 4.32 and 4.34 where $\overline{d}$ is a list of the variables $d_{p,a,q}$, $\overline{f}$ is a list of the variables f_q, and $\overline{y}$ is a list of the variables $y_{u,q}$ for $p, q \in Q$, $a \in \Sigma$, and $u \in C$. Moreover, let $\varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f})$ be as in Section 4.2.1. Then, we obtain the following result.

Lemma 4.14. *If* $\mathfrak{M} \models \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{C}}(\overline{d}, \overline{f}, \overline{y})$*, then* $u \in L(\mathcal{A}_\mathfrak{M})$ *implies* $u' \in L(\mathcal{A}_\mathfrak{M})$ *for all* $(u, u') \in \mathcal{C}$*.*

Proof. Let $\mathfrak{M} \models \varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{C}}(\overline{d}, \overline{f}, \overline{y})$. An induction analogous to the one in the correctness proof of Heule and Verwer's passive learning method (see Section 3.1.4) using Formulas 4.30 to 4.32 shows that $\mathcal{A}_\mathfrak{M} \colon q_0 \xrightarrow{u} q$ implies $y_{u,q}^\mathfrak{M} = \mathit{true}$ for all $u \in \mathrm{Pref}(C)$.

In order to prove that $\mathcal{A}_\mathfrak{M}$ respects the implications in $\mathcal{C}$, choose an implication $(u, u') \in \mathcal{C}$, and suppose $u \in L(\mathcal{A}_\mathfrak{M})$. The latter assumption means that $\mathcal{A}_\mathfrak{M} \colon q_0 \xrightarrow{u} p$ holds for some $p \in F$. This in turn implies $y_{u,p}^\mathfrak{M} = \mathit{true}$. Moreover, since $p \in F$ holds, we know that $f_p^\mathfrak{M} = \mathit{true}$ is satisfied by definition of $\mathcal{A}_\mathfrak{M}$.

On the other hand, $\mathcal{A}_\mathfrak{M} \colon q_0 \xrightarrow{u'} q$ implies $y_{u',q}^\mathfrak{M} = \mathit{true}$. Furthermore, Formula 4.34 then enforces that f_q must also be set to *true*. This means that $q \in F$ holds by definition of $\mathcal{A}_\mathfrak{M}$ and, therefore, $u' \in L(\mathcal{A}_\mathfrak{M})$. Thus, $u \in L(\mathcal{A}_\mathfrak{M})$ implies $u' \in L(\mathcal{A}_\mathfrak{M})$ for all $(u, u') \in \mathcal{C}$ as $(u'u') \in \mathcal{C}$ was chosen arbitrarily. □

It is left to express that the prospective DFA is consistent with the sample $\mathcal{S}$. To this end, we simply reuse the formula $\varphi_n^\mathcal{S}$ in Propositional Boolean Logic of Section 4.3.2. Then, $\varphi_n^{\mathcal{S},\mathcal{C}}$ is the conjunction

$$\varphi_n^{\mathcal{S},\mathcal{C}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}) := \varphi_n^{\mathrm{DFA}}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d}, \overline{f}, \overline{x}) \wedge \varphi_n^{\mathcal{C}}(\overline{d}, \overline{f}, \overline{y}),$$

and we obtain the following result.

Lemma 4.15. *Let* $\mathcal{S} = (S_+, S_-)$ *be a sample over* Σ*,* $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ *a finite set of implications,* $n \in \mathbb{N}_+$*, and*

$$\varphi_n^{\mathcal{S},\mathcal{C}}(\overline{d}, \overline{f}, \overline{x}, \overline{y}) := \varphi_n^{DFA}(\overline{d}, \overline{f}) \wedge \varphi_n^{\mathcal{S}}(\overline{d}, \overline{f}, \overline{x}) \wedge \varphi_n^{\mathcal{C}}(\overline{d}, \overline{f}, \overline{y}).$$

Then, the following statements hold:

1. *If* $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}(\overline{d}, \overline{f}, \overline{x}, \overline{y})$*, then* $\mathcal{A}_\mathfrak{M}$ *is a DFA with n states that satisfies* $S_+ \subseteq L(\mathcal{A}_\mathfrak{M})$*,* $S_- \cap L(\mathcal{A}_\mathfrak{M}) = \emptyset$*, and* $u \in L(\mathcal{A}_\mathfrak{M})$ *implies* $u' \in L(\mathcal{A}_\mathfrak{M})$ *for all* $(u, u') \in \mathcal{C}$*.*

2. *If a DFA* $\mathcal{A}$ *with n states exists that satisfies* $S_+ \subseteq L(\mathcal{A})$*,* $S_- \cap L(\mathcal{A}) = \emptyset$*, and* $u \in L(\mathcal{A}_\mathfrak{M})$ *implies* $u' \in L(\mathcal{A}_\mathfrak{M})$ *for all* $(u, u') \in \mathcal{C}$*, then* $\varphi_n^{\mathcal{S},\mathcal{C}}(\overline{d}, \overline{f}, \overline{x}, \overline{y})$ *is satisfiable.*

Lemma 4.15 follows from Lemma 4.14 and Lemma 4.8 (on Page 106). In addition, we obtain the following remark about the size of $\varphi_n^{\mathcal{S},\mathcal{C}}$.

Remark 4.6. In conjunctive normal form, the formula $\varphi_n^{\mathcal{S},\mathcal{C}}(\overline{d}, \overline{f}, \overline{x}, \overline{y})$ ranges over

$$\mathcal{O}\Big(n^2|\Sigma| + n|\mathrm{Pref}(S_+ \cup S_-)| + n|\mathrm{Pref}(C)|\Big)$$

variables and comprises

$$\mathcal{O}\Big(n^3|\Sigma| + n^2|\mathrm{Pref}(S_+ \cup S_-)| + n^2|\mathrm{Pref}(C)|\Big)$$

clauses.

Implementing $\varphi_n^{\mathcal{S},\mathcal{C}}$ in the Logic of Uninterpreted Functions over the Natural Numbers

To implement $\varphi_n^{\mathcal{S},\mathcal{C}}$ in the logic of uninterpreted functions over the natural numbers, we reuse the corresponding formulas φ_n^{DFA} and $\varphi_n^{\mathcal{S}}$ of Section 4.3.2. Moreover, we translate the formula $\varphi_n^{\mathcal{C}}$ in Propositional Boolean Logic into the logic of uninterpreted functions over $\mathbb{N}$.

To simplify this translation, we assume without loss of generality that all words in $\mathrm{Pref}(C)$ are enumerated, say $\mathrm{Pref}(C) = \{u_0, \ldots, u_{k-1}\}$ with $u_0 = \varepsilon$. Then, $\varphi_n^{\mathcal{S},\mathcal{C}}$ is the conjunction of Formulas 4.35 to 4.37, which are shown next.

$$y(0) = 0 \tag{4.35}$$

$$\bigwedge_{u_i, u_j \in \mathrm{Pref}(C), a \in \Sigma, u_j = u_i a} y(j) = d(y(i), a) \tag{4.36}$$

$$\bigwedge_{(u_i, u_j) \in \mathcal{C}} f(y(u_i)) \neq 0 \rightarrow f(y(u_j)) \neq 0 \tag{4.37}$$

Here, $d\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $f\colon \mathbb{N} \to \mathbb{N}$, and $y\colon \mathbb{N} \to \mathbb{N}$ are uninterpreted functions that correspond to the Boolean variables $d_{p,a,q}$, f_q, and $y_{u,q}$, respectively.

Finally, let $\varphi_n^{\mathcal{S},\mathcal{C}}$ be the conjunction

$$\varphi_n^{\mathcal{S},\mathcal{C}}(d,f,x,y) \coloneqq \varphi_n^{\mathrm{DFA}}(d,f) \land \varphi_n^{\mathcal{S}}(d,f,x) \land \varphi_n^{\mathcal{C}}(d,f,y).$$

Then, we obtain the following result.

Lemma 4.16. *Let $\mathcal{S} = (S_+, S_-)$ be a sample over Σ, $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ a finite set of implications, $n \in \mathbb{N}_+$, and*

$$\varphi_n^{\mathcal{S},\mathcal{C}}(d,f,x,y) \coloneqq \varphi_n^{DFA}(d,f) \land \varphi_n^{\mathcal{S}}(d,f,x) \land \varphi_n^{\mathcal{C}}(d,f,y).$$

Then, the following statements hold:

1. *If $\mathfrak{M} \models \varphi_n^{\mathcal{S},\mathcal{T}}(d,f,x,y)$, then $\mathcal{A}_{\mathfrak{M}}$ is a DFA with n states that satisfies $S_+ \subseteq L(\mathcal{A}_{\mathfrak{M}})$, $S_- \cap L(\mathcal{A}_{\mathfrak{M}}) = \emptyset$, and $u \in L(\mathcal{A}_{\mathfrak{M}})$ implies $u' \in L(\mathcal{A}_{\mathfrak{M}})$ for all $(u,u') \in \mathcal{C}$.*

2. *If a DFA $\mathcal{A}$ with n states exists that satisfies $S_+ \subseteq L(\mathcal{A})$, $S_- \cap L(\mathcal{A}) = \emptyset$, and $u \in L(\mathcal{A}_{\mathfrak{M}})$ implies $u' \in L(\mathcal{A}_{\mathfrak{M}})$ for all $(u,u') \in \mathcal{C}$, then $\varphi_n^{\mathcal{S},\mathcal{C}}(d,f,x,y)$ is satisfiable.*

A proof is again straightforward using the individual properties of the subformulas of $\varphi_n^{\mathcal{S},\mathcal{C}}$. Moreover, we obtain the following remark about the size of $\varphi_n^{\mathcal{S},\mathcal{C}}$.

Remark 4.7. The formula $\varphi_n^{\mathcal{S},\mathcal{C}}(d,f,x,y)$ ranges over four uninterpreted functions and comprises

$$\mathcal{O}\Big(n|\Sigma| + |\mathrm{Pref}(S_+ \cup S_-)| + |\mathrm{Pref}(C)|\Big)$$

clauses.

4.4.3 CEGAR-Style ICE-learner for Regular Model Checking

The *CEGAR-style ICE-learner* is an adaptation of the CEGAR-style learner (described in Section 4.3.3) to the ICE-learning setting. In addition to a sample $\mathcal{S} = (S_+, S_-)$ containing two finite sets $S_+, S_- \subseteq \Sigma^*$, the CEGAR-style ICE-learner maintains a finite set $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ of implications. In every iteration, the learner computes a minimal DFA $\mathcal{A}$ that is consistent with $\mathcal{S}$ and respects the implications in $\mathcal{C}$ using Algorithm 4.5 (see Page 116) and poses an equivalence query with $\mathcal{A}$. If the teacher replies "yes", the learning terminates. If the teacher returns a counterexample $u \in \Sigma^*$, the learner adds u to S_+ if $u \notin L(\mathcal{A})$, respectively to S_- if $u \in L(\mathcal{A})$. If the teacher returns an implication-counterexample (u,u'), the learner simply adds (u,u') to $\mathcal{C}$. Then, the learner proceeds with the next iteration. Algorithm 4.6 presents the complete procedure in pseudo code.

The following theorem states the correctness of Algorithm 4.6.

Theorem 4.17. *Let $\mathcal{P} = (I,T)$ be a program over the common alphabet Σ and $B \subseteq \Sigma^*$ a regular set of bad configurations with $B \cap I = \emptyset$. Moreover, let an ICE-teacher for $\mathcal{P}$ and B be given. If an IDFA with respect to $\mathcal{P}$ and B exists, Algorithm 4.6 terminates and returns a smallest IDFA.*

A proof of can easily be assembled from the correctness proof of the CEGAR-style learner (see Theorem 4.11 on Page 108) and the proof of Lemma 4.13 (on Page 116).

4.4.4 Angluin-Style ICE-learner for Regular Model Checking

The Angluin-style ICE-learner lifts the concept of the Angluin-style learner from Section 4.3 to the setting of learning from ICE-teachers. As before, the Angluin-style ICE-learner maintains an extended observation table $O_? = (R,S,T_?)$ in which it records answers to membership queries and counterexamples. Moreover, it additionally uses a set $\mathcal{C} \subseteq \Sigma^* \times \Sigma^*$ to store implication-counterexamples like the CEGAR-style ICE-learner.

Algorithm 4.6: CEGAR-style ICE-learner for Regular Model Checking.

Input: An ICE-teacher for Regular Model Checking.

```
1  Initialize a sample 𝒮 = (S+, S−) with S+ = ∅ and S− = ∅.
2  Let 𝒞 = ∅.
3  repeat
4  |  Compute a smallest DFA 𝒜 that is consistent with 𝒮 respects the implications
   |  in 𝒞 using Algorithm 4.5.
5  |  Perform an equivalence query with 𝒜.
6  |  if the teacher returns a counterexample u then
7  |  |  if u ∈ L(𝒜) then
8  |  |  |  S− ← S− ∪ {u}.
9  |  |  else
10 |  |  |  S+ ← S+ ∪ {u}.
11 |  |  end
12 |  else if the teacher returns an implication-counterexample (u, u′) then
13 |  |  𝒞 ← 𝒞 ∪ {(u, u′)}.
14 |  end
15 until the teacher replies "yes" to the equivalence query with 𝒜.
16 return 𝒜.
```

Algorithm 4.7 (on Page 122) presents the Angluin-style ICE-learner in pseudo code. The learner starts with an initial extended observation table $O_?$ and an empty set $\mathcal{C}$. In every iteration, the learner makes the table weakly closed and weakly consistent in the same manner as the Angluin-style learner of Section 4.3.4. Once this is the case, it derives a sample $\mathcal{S} = (S_+, S_-)$ from the table where

$$S_+ = \{u \mid T_?(u) = 1\} \text{ and } S_- = \{u \mid T_?(u) = 0\}.$$

Given this sample, the learner invokes Algorithm 4.5 (see Page 116) to compute a minimal DFA $\mathcal{A}$ that is consistent with $\mathcal{S}$ and respects the implications in $\mathcal{C}$. Then, the learner poses an equivalence query with $\mathcal{A}$. If the teacher replies "yes", the learner halts and outputs $\mathcal{A}$. Otherwise, the teacher either replies a counterexample u or an implication counterexample (u, u'). In the first case, the learner adds u and all of its prefixes to R and updates the table. In the latter case, the learner adds (u, u') to $\mathcal{C}$. Then, the learner proceeds with the next iteration.

The next theorem states the correctness of Algorithm 4.7.

Theorem 4.18. *Let $\mathcal{P} = (I, T)$ be a program over the common alphabet Σ and $B \subseteq \Sigma^*$ a regular set of bad configurations with $B \cap I = \emptyset$. Moreover, let an ICE-teacher for $\mathcal{P}$ and B be given. If an IDFA with respect to $\mathcal{P}$ and B exists, Algorithm 4.7 terminates and returns a smallest IDFA.*

Algorithm 4.7: Angluin-style ICE-learner for Regular Model Checking.

Input: An ICE-teacher for Regular Model Checking.

1 Initialize the extended observation table $O_? = (R, S, T_?)$ with $R = S = \{\varepsilon\}$ and *update*$(O_?)$.
2 Let $\mathcal{C} = \emptyset$.
3 **repeat**
4 Produce a weakly closed and weakly consistent extended observation table using membership queries.
5 Let $\mathcal{S} = (\{u \mid T_?(u) = 1\}, \{u \mid T_?(u) = 0\})$.
6 Compute a smallest DFA $\mathcal{A}$ consistent with $\mathcal{S}$ that respects the implications in $\mathcal{C}$ using Algorithm 4.5.
7 Perform an equivalence query with $\mathcal{A}$.
8 **if** *the teacher returns a counterexample u* **then**
9 $R \leftarrow R \cup \mathrm{Pref}(u)$.
10 *update*$(O_?)$.
11 **else if** *the teacher returns an implication-counterexample (u, u')* **then**
12 $\mathcal{C} \leftarrow \mathcal{C} \cup \{(u, u')\}$;
13 **end**
14 **until** *the teacher replies "yes" to the equivalence query with $\mathcal{A}$.*
15 **return** $\mathcal{A}$.

As in the case of the CEGAR-style ICE-learner, a proof of Theorem 4.18 can easily be assembled from the correctness proof of the Angluin-style learner (see Theorem 4.12 on Page 111) and the proof of Lemma 4.13 (on Page 116).

4.5 Experiments and Evaluation

This section consists of two parts. In the first part, we briefly highlight the differences between our algorithms and the tools T(o)rmc, Faster, and Lever from a user's perspective; that is, we discuss differences in the input formats as well as what knowledge the user needs. In the second part, we assess the performance of our algorithms (in terms of runtime) based on a prototype implementation and compare this implementation to T(o)rmc and Faster.

Differences to Existing Tools

We already discussed the tools T(o)rmc, Lever, and Faster in the section about related work. Here we highlight their differences to the techniques developed in this chapter.

Differences to T(o)rmc T(o)rmc [Leg08] implements a white-box algorithm that iterates the given transducer on the set of initial configurations and applies extrapolation to approximate the limit of the iteration. The drawback of this method is that the bad configurations are not taken into account during the computation; if the result contains a bad configuration, T(o)rmc has to be restarted with additional user input, which requires expert knowledge about both T(o)rmc's internals and the problem at hand. Another drawback of T(o)rmc is that it requires DFAs as input, whereas our algorithms also work with NFAs, which can be exponentially smaller than equivalent DFAs. Also, T(o)rmc does not search for a smallest invariant, whereas our algorithms do.

Differences to Lever The Lever tool [VV06] implements a learning-based black-box algorithm that builds upon Kearns and Vazirani's learning algorithm. When used for Regular Model Checking (recall that Lever can also be used for the verification of liveness properties), Lever tries to learn a fixed point representing the exact set of reachable configurations. However, Lever does not learn this set directly but a set of configuration-witness pairs, which consist of a configuration augmented with "distance information". Compared to the learning-based algorithms of this chapter, Lever's approach has the advantage that the set of configuration-witness pairs is unique—whereas we aim for an arbitrary invariant, of which there might be many. The uniqueness of the target language makes answering membership and equivalence queries possible and permits a straightforward application of standard active learning algorithms.

In order to learn sets of configurations-witness pairs, Lever requires an encoding that translates such pairs into finite words and vice versa. However, finding a suitable encoding requires expert knowledge about both Lever and the given problem domain. Another limiting factor of Lever is that a minimal DFA accepting the set of configuration-witness pairs can be larger than a minimal IDFA because the former needs to represent more information. This, however, is a crucial aspect since the runtime of Lever (like most learning-based algorithms) depends on the size of the learned automaton.

Differences to Faster Faster [BLP06] computes the exact set of reachable configurations using acceleration. This approach prevents Faster from terminating if the set of reachable configurations is not recognizable by a finite automaton. In contrast, our algorithms always find an IDFA if one exists. Moreover, Faster was originally designed for integer linear systems over Presburger formulas. That entails that one first has to translate a given Regular Model Checking instance into the Faster input

Table 4.1: Feature summary of algorithms for Regular Model Checking.

	T(O)RMC	FASTER	LEVER	White-box	Semi-black-box	Black-box
Mode	White-box	White-box	Black-box	White-box	Semi-black-box	Black-box
Input	DFAs	DFA and formulas	Teacher	NFAs	NFAs and teacher	Teacher
Experience	Expert	Expert	None	None	None	None
Target concept	Invariant	Reachable config.	Conf.-witn. pairs	Invariant	Invariant	Invariant
Minimality	no	no	no	yes	yes	yes

format, which requires manual work as a translation is often not straightforward (if it is possible at all).

In conclusion, Table 4.1 summarizes the main features of the considered algorithms for Regular Model Checking. The rows "Mode" and "Input" are self-explanatory. The row "Experience" refers to the question of whether the user needs any (expert) knowledge either about the program at hand or the internals of the algorithm. The row "Target concept" refers to the kind of concept the respective algorithm computes. Finally, the row "Minimality" indicates whether an algorithm searches for a smallest representation of the target concept.

Experimental Results

To assess the effectiveness and performance of our algorithms, we implemented a prototype and benchmarked it against FASTER and T(O)RMC. Due to the fact that LEVER is not publicly available, a comparison to this tool was not possible. The results of this section partly appeared in conference proceedings [Nei12].

Methodology We implemented our prototype in C++ using AMoRE++ [KMP+89] as a backend for operations on automata and LIBALF for learning automata. As underlying logic solvers, we used GLUCOSER (for solving SAT formulas) and Microsoft's Z3 (for solving formulas with uninterpreted functions).

We considered two benchmark suites. The first benchmark suite contains *integer linear systems* (mostly protocols, such as the Berkeley cache coherence protocol, the Synapse cache coherence protocol, and the M.E.S.I. cache coherence protocol) and is available on the FASTER and T(O)RMC websites[7]; additionally, we added three petri nets (*trans1*, *trans2*, and *trans3*). The second benchmark suite contains instances of a 2^n modulo-counter and the token ring protocol (see Example 4.1 on Page 91) over a fixed

[7] http://www.lsv.ens-cachan.fr/Software/fast/examples/examples.tgz

number of processes. In the second benchmark suite, we successively enlarged the input-automata $\mathcal{A}^I$ and $\mathcal{A}^B$, with the motivation to demonstrate the advantages of our semi-black-box and black-box algorithms when confronted with large input-automata. Note, however, that we did not vary the size of the transducer. We comment on this decision on shortly when discussing the results of our experiments.

The examples of the second benchmark suite were not natively expressible as Faster inputs. In order to avoid a biased benchmark, we decided not to run Faster on the second benchmark suite.

Compiling T(o)rmc for 64-bit systems did not work properly. A 32-bit executable partly worked but suffered from memory access violations. We experienced crashes, and it was often not possible to conduct experiments; for instance, we could not obtain any result for the modulo counter experiments because T(o)rmc crashed on all inputs. We contacted the tool's developer, but the problem could not be resolved so far. Thus, we can report T(o)rmc's results only for a part of the experiments.

We conducted all experiments on an Intel Q9550 CPU at 2.83 GHz with 4 GiB of RAM running Ubuntu 12.04 LTS. We imposed a timeout limit of 300 s.

Results Tables 4.2 to 4.4 (on Pages 126 and 127) present the results of our experiments. All runtimes in the tables are in seconds. A "—" indicates that the corresponding experiment either ran out of memory or did not finish within 300 s. An "x" means that the experiment crashed. The best result of each experiment is highlighted in bold font.

Table 4.2 shows the results on integer linear systems of the first benchmark suite. The white-box algorithm using Glucoser often performed best, closely followed by Faster. The only exception is the bakery protocol, which none of our algorithms could prove correct. However, the performance of all algorithms is relatively similar on this benchmark suite, and no algorithm excelled. The algorithms using Glucoser performed slightly better than those using Z3.

Tables 4.3 and 4.4 show the results on the second benchmark suite; Table 4.3 reports the results on the modulo counter experiments, and Table 4.4 reports the results on the token ring experiments.

In the case of the modulo counter experiments, the algorithms again performed similarly. The semi-black-box approach using the Angluin-style learner and Glucoser achieved the best results, closely followed by the semi-black-box approach using the CEGAR-style learner and Glucoser. Again, the algorithms using Glucoser performed slightly better than those using Z3. Unfortunately, T(o)rmc did not produce any results on this examples.

In the case of the token ring experiments, the white-box algorithm outperformed any other algorithm, regardless of the underlying logic solver. The Angluin-style

Table 4.2: Results on integer linear systems of the first benchmark suite. All figures are in seconds. A "—" corresponds to a timeout after 300 s. An "x" indicates that the experiment crashed. The best result of each experiment is highlighted in bold font.

Experiment	White-box		Semi-black-box				Black-box				T(O)RMC	Faster
			Angluin		CEGAR		Angluin		CEGAR			
	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3		
petri net	**0.01**	0.05	0.12	0.15	0.11	0.11	0.70	0.96	0.07	0.23	0.02	1.13
berkeley	0.04	0.41	0.62	0.92	1.29	1.45	1.80	1.81	1.79	1.55	4.23	**0.03**
synapse	**0.01**	0.03	0.04	0.07	0.06	0.16	0.02	0.07	0.02	0.11	0.19	0.03
lift	**0.01**	0.14	**0.01**	**0.01**	**0.01**	0.02	0.12	0.13	0.12	0.12	5.54	0.15
mesi	0.45	1.78	0.58	2.64	1.55	6.24	26.42	52.13	27.93	47.48	5.52	**0.04**
bakery	—	—	—	—	—	—	—	—	—	—	32.18	**0.04**
trans1	**0.01**	0.03	**0.01**	0.05	0.02	0.18	0.02	0.17	0.02	0.18	x	0.04
trans2	**0.01**	0.03	**0.01**	0.06	0.02	0.05	0.03	0.14	0.02	0.16	x	0.03
trans3	**0.04**	0.27	0.05	0.29	0.09	0.53	3.13	6.33	2.36	2.88	x	0.07

Table 4.3: Results on modulo counter experiments. All figures, except for those in the columns "$|\mathcal{A}^I|$" and "$|\mathcal{A}^B|$", are in seconds. A "—" corresponds to a timeout after 300 s. The best result of each experiment is highlighted in bold font.

$\lvert\mathcal{A}^I\rvert$	$\lvert\mathcal{A}^B\rvert$	White-box		Semi-black-box				Black-box			
				Angluin		CEGAR		Angluin		CEGAR	
		Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3
14	125	0.24	0.46	0.29	0.41	0.75	1.03	**0.17**	0.37	1.59	2.73
14	156	**0.29**	1.33	0.58	0.99	1.75	2.09	0.34	0.65	3.33	6.87
34	187	1.29	8.29	**1.13**	3.52	4.04	6.48	1.17	6.12	9.11	29.30
34	218	27.49	64.29	**2.49**	20.42	6.45	47.84	5.95	33.13	35.16	80.14
82	249	—	—	**21.27**	100.48	45.23	178.59	—	177.92	—	—

Table 4.4: Results on token ring experiments. All figures, except for those in the columns "$|\mathcal{A}^I|$" and "$|\mathcal{A}^B|$", are in seconds. A "—" corresponds to a timeout after 300 s. The best result of each experiment is highlighted in bold font.

$\lvert\mathcal{A}^I\rvert$	$\lvert\mathcal{A}^B\rvert$	White-box		Semi-black-box				Black-box				T(o)rmc
				Angluin		CEGAR		Angluin		CEGAR		
		Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	Glucoser	Z3	
10	3	**0.01**	**0.01**	0.02	0.07	0.02	0.07	0.04	0.04	**0.01**	0.10	0.02
25	3	0.03	**0.02**	0.12	0.21	0.03	0.10	0.18	0.18	**0.02**	0.10	0.06
50	3	**0.02**	**0.02**	1.23	1.52	0.07	0.14	1.22	1.45	0.05	0.14	0.31
100	3	0.04	**0.02**	21.60	23.39	0.31	0.47	20.89	22.58	0.33	0.48	2.08
200	3	**0.04**	**0.04**	—	—	2.38	1.84	—	—	2.15	2.50	16.13
300	3	**0.04**	0.05	—	—	7.10	5.82	—	—	8.53	8.70	55.13
400	3	**0.04**	0.07	—	—	18.75	15.55	—	—	18.66	20.70	137.47
500	3	**0.05**	0.09	—	—	31.26	27.97	—	—	38.26	38.54	290.41

learner and the Angluin-style ICE-learner performed worst and failed on all instances with $|\mathcal{A}^I| > 100$. T(o)rmc succeeded in all cases, but was slowest among all successful algorithms on instances with $|\mathcal{A}^I| > 100$. In contrast to the experiments above, we did not observe a difference in the performance between algorithms using Glucoser and algorithms using Z3.

Discussion Considering the results of our experiments, we make two key observations. First, the results of the first benchmark suite show that all of our algorithms can handle problem instances specified for T(o)rmc and Faster with competitive runtimes. Second, we observe that there is no superior algorithm. The white-box algorithm often performs best, but the modulo counter examples show that learning-based algorithms can be advantageous in situations where the input-automata are large (cf. Table 4.3). Moreover, the second benchmark suite contains examples on which the Angluin-based algorithms outperformed the CEGAR-based algorithms (cf. Table 4.3) and vice versa (cf. Table 4.4).

For the benchmarks at hand, the algorithms using the Glucoser SAT solver were always slightly faster than the ones using the Z3 SMT solver. Note, however, that this might be different for larger instances as the size of the generated SAT formulas grows faster than the size of the SMT formulas. A further observation is that Z3 seemed to produce an initialization overhead every time it was invoked, which constituted a large share of the overall runtime on small instances.

Finally, let us comment on our decision to only consider experiments in which we varied $\mathcal{A}^I$ and $\mathcal{A}^B$ but not the transducer. We observed that the black-box approach spend a large share of its time on checking whether a conjecture is inductive, and it turned out that AMoRE++ is not well-suited for this task[8]. Since this is a problem of AMoRE++ but not of our black-box algorithm (an ICE-teacher completely abstracts from an actual implementation), benchmarking the present prototype on experiments with varying transducers makes it very hard to draw any conclusions on the performance of our black-box algorithms. However, we expect more meaningful results from using a different automata library.

4.6 Further Applications

One can use the algorithms described in this chapter for more than computing invariants in the context of Regular Model Checking. We demonstrate this claim in the next two subsections. In Section 4.6.1, we consider the problem of finding so-called *minimal separating DFAs*. In Section 4.6.2, we demonstrate how Regular Model Check-

[8] AMoRE++ was mainly designed to translate between automata, monoids, and regular expressions.

ing can be used to synthesize loop invariants of WHILE programs in the context of Floyd-Hoare-style verification.

4.6.1 Computing Minimal Separating DFAs

A separating DFA is defined with respect to two disjoint regular languages $L_1, L_2 \subseteq \Sigma^*$. Formally, a DFA $\mathcal{A}$ satisfying $L_1 \subseteq L(\mathcal{A})$ and $L_2 \cap L(\mathcal{A}) = \emptyset$ is said to be a *separating DFA*, and a *minimal separating DFA* is a separating DFA of minimal size. Note that minimal separating DFAs are not unique for fixed L_1, L_2 since their behavior on words not belonging to L_1 or L_2 is unspecified. In fact, computing a minimal separating DFA for two disjoint regular languages is computationally hard because the corresponding decision problem

> "Given two disjoint regular languages $L_1, L_2 \subseteq \Sigma^*$ and $k \in \mathbb{N}_+$. Does a separating DFA with k states exist?"

is NP-complete (e.g., see Pfleeger [Pfl73]). Thus, applying solver-based algorithms to this task is reasonable.

Minimal separating DFAs are helpful in various contexts. For instance, a direct application to compositional verification is described by Chen et al. [CFC+09]. Another example is the well-known task of minimizing incompletely specified DFAs [PO98], which can also be phrased in terms of minimal separating DFAs. An incompletely specified DFA is a DFA $\mathcal{B} = (Q, \Sigma, q_0, \delta, \mathrm{Acc}, \mathrm{Rej}, \mathrm{Unspec})$ whose states are partitioned into accepting, rejecting, and unspecified states. The task of minimizing an incompletely specified DFA now is to compute a DFA of minimal size that accepts all words that lead to an accepting state in $\mathcal{B}$ and rejects all words that lead to a rejecting state in $\mathcal{B}$ (while ignoring all remaining words). This, however, is the same as computing a minimal separating DFA for the languages

$$L_1 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Acc}\} \text{ and } L_2 = \{u \in \Sigma^* \mid \mathcal{B}\colon q_0 \xrightarrow{u} q, q \in \mathrm{Rej}\}.$$

As a final example, let us mention passive learning from a sample $\mathcal{S} = (S_+, S_-)$: the passive learning task amounts to finding a minimal separating DFA for the finite languages $L_1 = S_+$ and $L_2 = S_-$.

Roughly speaking, a (minimal) separating DFA is an invariant in the sense of Section 4.1 except for the inductivity constraint. Thus, we can easily apply the algorithms of Sections 4.2 to 4.4 with the following input:

- An NFA $\mathcal{A}^I$ accepting the language L_1
- An NFA $\mathcal{A}^B$ accepting the language L_2

```
Input: x

r = 0;
y = x;
while(y > 0) {
        r = r + 3;
        y = y - 1;
}
```

(a) An While program.

$$
\begin{aligned}
G(x,y,r) &\coloneqq y > 0 \\
\varphi_{\text{pre}}(x,y,r) &\coloneqq r = 0 \land y = x \\
\varphi_{\text{post}}(x,y,r) &\coloneqq y = 0 \land \\
&\quad\; \exists t\colon x + x = t \land x + t = r \\
\varphi_{\text{loop}}(x,y,r,x',y',r') &\coloneqq x' = x \land y' = y - 1 \land \\
&\quad\; r' = r + 3
\end{aligned}
$$

(b) Presburger formulas annotating the While loop of the program in Figure 4.3a.

Figure 4.3: A While program and Presburger formulas annotating the While loop.

- An asynchronous transducer $\mathcal{T}$ accepting the identity relation

An alternative way to apply our solver-based algorithms of Sections 4.2 and 4.3 would be to alter the logic formulas and to drop the subforumla $\varphi_n^{\mathcal{T}}$ because $\varphi_n^{\mathcal{T}}$ is a tautology in the case that $\mathcal{T}$ defines the identity relation. For instance, it is sufficient for the white-box algorithm of Section 4.2 to use the formula

$$\varphi_n^{\text{DFA}} \land \varphi_n^{\mathcal{A}^I} \land \varphi_n^{\mathcal{A}^B}.$$

In fact, this example shows that our algorithms for Regular Model Checking can easily be adapted to new settings by adding—or removing—logic subformulas. We believe that this versatility is a significant advantage of our techniques, which hopefully makes our approach interesting for other researchers as well.

4.6.2 Synthesizing Presburger Loop Invariants

Next, we consider Floyd-Hoare-style verification [Flo67, Hoa69] in the context of While *programs* working over integer variables. We do not want to give a formal definition of such programs here but rather use an example to illustrate what a prototypical program looks like.

Example 4.3. An example of a While program is depicted in Figure 4.3a. The program takes a variable x as input and computes $3x$. The result of this computation is stored in the variable r.

One of the most important challenges in Floyd-Hoare-style verification is to automatically synthesize loop invariants. Here, we are interested in the special case of synthesizing invariants for While *loops* that are annotated in Presburger arithmetic. More precisely, we assume the following setting:

- Sets of program configurations are represented by Presburger formulas $\varphi(\overline{x})$ ranging over the program variables $\overline{x} = (x_1, \ldots, x_n)$ where each variable assumes a value in $\mathbb{Z}$.
- The loop is annotated with a *precondition* $\varphi_{\text{pre}}(\overline{x})$, a *postcondition* $\varphi_{\text{post}}(\overline{x})$, and a formula $\varphi_{\text{loop}}(\overline{x}, \overline{x}')$ that describes the effect of the loop on the program variables; in the latter formula, $\overline{x}$ corresponds to the variables before the loop body is executed, whereas $\overline{x}'$ corresponds to the (potentially altered) program variables after the loop body has been executed. Moreover, $G(\overline{x})$ is the *guard* of the loop.

Figure 4.3b shows an annotation of the While loop in Figure 4.3a.

Broadly speaking, a loop invariant is a statement about the configurations of a program that is true before and after every iteration of the loop. Formally, a *loop invariant* is a set Inv of program configurations that satisfies the following three properties:

- $\varphi_{\text{pre}}(\overline{x}) \rightarrow \overline{x} \in Inv$.
- $(\overline{x} \in Inv \wedge \neg G(\overline{x})) \rightarrow \varphi_{\text{post}}(\overline{x})$.
- $(\overline{x} \in Inv \wedge G(\overline{x}) \wedge \varphi_{\text{loop}}(\overline{x}, \overline{x}')) \rightarrow \overline{x}' \in Inv$.

In Example 4.3, the set of program configurations satisfying $3(x - y) = r$ is a loop invariant. In fact, this loop invariant is exactly what our technique, which we introduce shortly, computes for this example.

In order to apply our Regular Model Checking techniques, we exploit the connection between Presburger arithmetic and automata. More precisely, we turn the formulas φ_{pre}, φ_{post}, and G into DFAs $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{\varphi_{\text{post}}}$, and $\mathcal{A}^G$ working over the alphabet $\Sigma = \{0, 1\}^n$, and φ_{loop} into an asynchronous transducer $\mathcal{T}^{\varphi_{\text{loop}}}$ working over the input and output alphabet Σ.[9] Then, we can reformulate the definition of loop invariants from above such that a loop invariant now is a set $Inv \subseteq \Sigma^*$ matching the encoding of program configurations used by $\mathcal{A}^{\varphi_{\text{pre}}}$, $\mathcal{A}^{\varphi_{\text{post}}}$, $\mathcal{A}^G$, and $\mathcal{T}^{\varphi_{\text{loop}}}$ that satisfies

- $L(\mathcal{A}^{\varphi_{\text{pre}}}) \subseteq Inv$;
- $(\Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))) \cap Inv = \emptyset$
  (which is true if and only if $(Inv \cap (\Sigma^* \setminus L(\mathcal{A}^G))) \subseteq L(\mathcal{A}^{\varphi_{\text{post}}})$); and
- $(R(\mathcal{T}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))(Inv) \subseteq Inv$.

When phrased this way, and by setting $I = L(\mathcal{A}^{\varphi_{\text{pre}}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{\text{post}}}))$, and $T = (R(\mathcal{T}^{\varphi_{\text{loop}}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))$, synthesizing loop invariants amounts to computing invariants in Regular Model Checking. This immediately leads to the following result.

[9] See Section 2.2 for more details about the conversion from Presburger formulas to finite automata.

Theorem 4.19. *Let Presburger formulas φ_{pre}, φ_{post}, φ_{loop}, and G annotating the loop of a* While *program over n integer variables be given. Moreover, let $\Sigma = \{0,1\}^n$, $I = L(\mathcal{A}^{\varphi_{pre}})$, $B = \Sigma^* \setminus (L(\mathcal{A}^G) \cup L(\mathcal{A}^{\varphi_{post}}))$, and $T = (R(\mathcal{T}^{\varphi_{loop}}) \cap (L(\mathcal{A}^G) \times \Sigma^*))$. If a loop invariant in form of an IDFA exists, then the Regular Model Checking algorithms of Sections 4.2 to 4.4 with the input $\mathcal{P} = (I,T)$ and B terminate and return a smallest IDFA.*

Once we have found an IDFA, we can try to translate it into a Presburger formula (e.g., using the technique by Leroux [Ler05]). However, even if this is not possible, an IDFA is sufficient for Floyd-Hoare-style verification because it proves that the postcondition holds once the loop has terminated. Nonetheless, it would be interesting to investigate whether one can impose further constraints on the logic formulas used by our algorithms (corresponding to the characterization by Leroux) that guarantee that a learned IDFA can be translated into a Presburger formula.

To evaluate the approach described above, we extended the SAT-based implementation of Algorithm 4.1 with an interface to the Mona tool [HJJ+95]. Our implementation now takes Presburger formulas in the Mona syntax as input, calls Mona to translate the formulas into DFAs, and subsequently executes Algorithm 4.1 to search for an IDFA.

We benchmarked our prototype implementation on various example programs (mostly code fragments taken from real-world software) that are shipped with the InvGen toolkit [GR09]. About one tenth of these examples (nine in total) were suitable for our setting (i.e., they contained a While loop, and we could annotate the loop with Presburger formulas).

Table 4.5 presents our experimental results. The column "Size" displays the number of states of the resulting IDFA. The column "Presburger" indicates whether we could translate the resulting IDFA into a Presburger formula; since we did not use software for this task, a blank entry indicates that the learned IDFA did not obviously translate into a Presburger formula. As in Section 4.5, all experiments were performed on an Intel Q9550 CPU at 2.83 GHz with 4 GiB of running Ubuntu 12.04 LTS.

Each experiment was finished in less than 6 s and required at most 300 MiB of RAM. For all examples, our implementation identified a loop invariant and most of them could be translated into a formula in Presburger arithmetic. To put these results into context, InvGen also found loop invariants for all examples and used roughly the same amount of time and memory. This shows that our technique is competitive with InvGen on the considered examples.

Finally, we want to mention that this prototype is just a proof-of-concept and not meant to compete with mature tools such as InvGen. Rather, its purpose is to show how one can apply algorithms for Regular Model Checking to a different scenario as well as to demonstrate the versatility of our techniques.

Table 4.5: Experimental results of our prototype implementation on a selection of InvGen's "C test suite".

Experiment	Size	Time in s	Presburger
down.c	3	0.03	yes
gulwani_cegar2.c	3	0.05	yes
ken-imp.c	3	0.08	
NetBSD_g_Ctoc.c	2	0.05	yes
simple.c	2	0.02	yes
simple_if.c	2	0.02	yes
split.c	6	5.74	
up-nd.c	2	0.05	yes

4.7 Conclusion

We studied Regular Model Checking of safety properties and considered the task of synthesizing invariants. To tackle this task, we have developed several algorithms, ranging from a white-box algorithm, which is based on highly optimized SAT and SMT solver technology, on the one hand to learning-based black-box algorithms, which combine logic solvers with automata learning, on the other hand; in between lie semi-black-box algorithms, which abstract from the exact sets of initial and bad configurations but require access to the program's transducer.

We evaluated the performance of our algorithms based on a prototype implementation. Our prototype turned out to be competitive to Faster and T(o)rmc, especially when confronted with large input-automata. Moreover, our algorithms work out-of-the-box, do not require expert knowledge about their internals, and always find an invariant (in form of an IDFA) provided one exists.

Apart from Regular Model Checking, we demonstrated that our techniques can also be applied to other problems that amount to finding (smallest) automata with user-specified properties. In particular, we considered computing minimal separating DFAs as well as synthesizing loop invariants of While programs that are annotated with Presburger formulas. In fact, one can view our techniques as a generic toolkit from which an algorithm for a particular problem can be instantiated. We hope that such a toolkit comes in handy for other researchers and may be applied in different fields, too.

For future work, it would be interesting to investigate the applicability of our semi-black-box and black-box algorithms in situations in which the program, respectively the set of bad configurations, cannot be expressed in terms of finite automata, but where it is still possible to construct a suitable teacher. A natural starting point are

subclasses of context-free languages that have already been studied in the context of Regular Model Checking by Fisman and Pnueli [FP01]. The challenge of such extensions certainly is the development of software tools that allow constructing suitable teachers.

Another promising direction of further research would be to explore an incremental SAT approach in which clauses learned during the solving process are reused in order to avoid recurring restarts of the SAT solver.

A further opportunity for optimizations would be to use NFAs rather than DFAs as representations of invariants. Although a representation in terms of NFAs increases the size of the logic formulas, it potentially allows learning exponentially smaller automata. A further step in this direction would be to consider even more expressive automata models, such as realtime one-counter automata or visibly one-counter automata, for which learning algorithms are available [FR95, NL10].

Finally, let us come back to the problem of synthesizing loop invariants. As we have shown in Section 4.6.2, the idea of using automata learning for this task looks promising. However, we considered a rather restricted setting: the data type of program variables is restricted to integers, and annotations have to be in Presburger arithmetic. Although such a setting is often sufficient for hardware-related or low-level applications, today's software typically operates on the heap and relies on a complex interplay between data structures. To synthesize loop invariants for such sophisticated settings, we can no longer resort to Regular Model Checking but need to devise techniques that explicitly take characteristics of heap manipulating programs into account. We present such a technique in the next chapter.

5

Quantified Invariants of Linear Data Structures

In this chapter, we develop an active learning algorithm for *quantified logic formulas* describing invariants of programs manipulating linear data structures such as arrays and lists. Our precise aim is to develop an active learning algorithm that learns universally quantified first-order formulas of the form

$$\forall y_1 \colon \dots \forall y_k \colon \varphi(y_1, \dots, y_k),$$

where φ is quantifier-free and captures properties of arrays and lists; the variables allowed to occur in φ range over array indices, respectively list locations, and φ itself can refer to the data stored at these positions. We focus on properties that are expressible in the Array Property Fragment [BMS06] to formulate properties of programs manipulating arrays and the decidable syntactic fragment of Strand [MPQ11] to formulate properties of programs manipulating lists.[1]

As an example of the type of formulas we are interested in, consider the following Strand formula, which expresses a typical loop invariant in a sorting program over lists:

$$\forall y_1 \colon \forall y_2 \colon (\mathit{head} \rightarrow^* y_1 \land y_1 \rightarrow y_2 \land y_2 \rightarrow^* i) \rightarrow d(y_1) \leq d(y_2). \tag{5.1}$$

Thereby, $\rightarrow$ is the successor relation of list cells, $x \rightarrow^* y$ denotes that y points to a cell reachable from the cell pointed to by x, and $d(x)$ refers to the data stored in the cell pointed to by x. Formula 5.1 states the following: for every pair of cells pointed to by y_1 and y_2 that occur in the list between the cells referenced by the pointers *head* and i

[1] See Section 2.2 for an introduction to the Array Property Fragment and Strand.

and where y_2 points to the direct successor of y_1, the data stored at y_1 is less than or equal to the data stored at y_2. In other words, the sublist from *head* to i is sorted.

One can express the same property for a program manipulating arrays by the formula

$$\forall y_1 : \forall y_2 : (0 \leq y_1 \land y_2 = y_1 + 1 \land y_2 \leq i) \rightarrow a[y_1] \leq a[y_2], \tag{5.2}$$

where $a[i]$ refers to the array's data at index i.

Quantified data automata Our first contribution, which we present in Section 5.1, is a novel representation—a normal form—of universally quantified properties over linear data structures, called *quantified data automata (QDA)*. QDAs are the target concept of our learning algorithm and can be translated into logic formulas.

QDAs are based on modeling program configurations as finite words called *data words*, which combine the data stored in the heap with the program's pointer and primitive variables. In order to capture universally quantified properties, we extend data words with *valuations* of the universally quantified variables, which results in *valuation words*. Extending data words with valuations makes the universal quantification explicit.

As an example of valuation words, consider the program configuration of a fictitious program manipulating lists depicted in the upper part of Figure 5.1. The list consists of four cells, and the program uses the pointer variables $head, i, j$, and *tail* to reference cells of the list; the arrows indicate which list cell a pointer variable references. The variables y_1 and y_2 are universally quantified variables; in the current valuation, y_1 points to the second list cell, whereas y_2 points to the last. The valuation word corresponding to this program configuration is depicted in the lower part of Figure 5.1. We use two *blank-symbols*, namely b and $-$, to indicate that no pointer variable (indicated by b) or no universally quantified variable (indicated by $-$) appears in the corresponding component; moreover, we use the symbol $\underline{b} = (b, -)$ to denote positions at which no variable appears.

Intuitively, QDAs are register automata that are equipped with data formulas at their states. The semantic of a QDA is the following: it (deterministically) reads valuation words symbol-by-symbol and whenever it encounters a universally quantified variable, it stores the current data value in a register corresponding to this variable; it accepts a valuation word if its registers satisfy the data formula decorating the state finally reached. The universal quantification is captured by defining that a QDA accepts a data word if it accepts *all* possible valuation words that extend this data word. The key idea is that a QDA checks structural constraints of the data structure by means of its transition structure and constraints on the data by means of its data formulas.

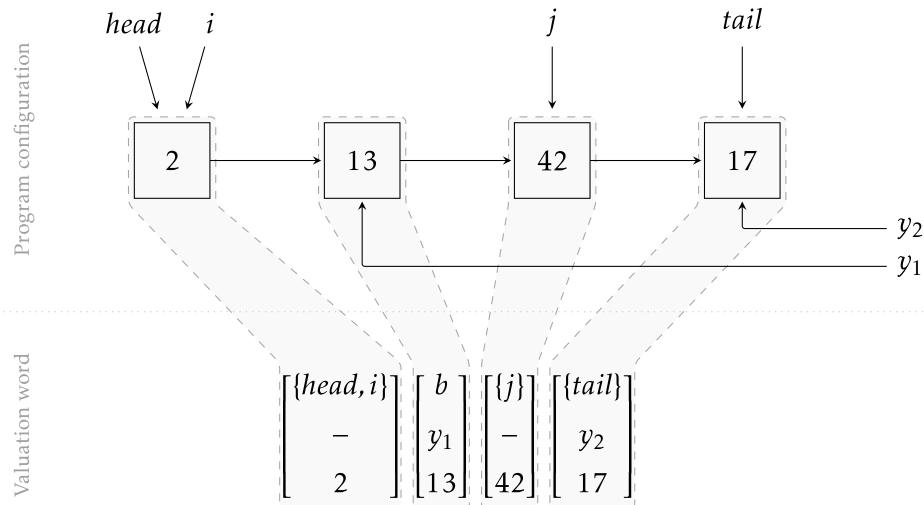


Figure 5.1: A program configuration of a fictitious program manipulating a list and the corresponding valuation word. The variables *head*, i, j, and *tail* are pointer variables referencing list cells, and y_1, y_2 are universally quantified variables (also referencing list cells).

An example QDA[2] is shown in Figure 5.2 (on Page 138). Missing transitions point to a sink-state labeled with the formula *false*, which is not shown in the figure for the sake of readability. When reading valuation words, the QDA in Figure 5.2 checks whether *head* occurs before or together with y_1, whether y_2 occurs immediately after y_1, and whether y_2 occurs before or together with i. If all of this is satisfied, it checks whether the data of the cell referenced by y_1 is less than or equal to the data of the cell referenced by y_2; otherwise, it simply accepts. The QDA, hence, accepts precisely the set of data words that correspond to program configurations satisfying Formula 5.1.

Active learning of quantified properties using QDAs In Section 5.2, we develop an efficient active learning algorithm for QDAs by combining abstraction over a set of data formulas and Angluin-style active learning algorithms. We first show that there exists a *canonical minimal* QDA for any set of valuation words. Using this result, we show that learning QDAs can be reduced to learning sets of so-called *formula words*—valuation words without data but paired with a data formula—which in turn can be reduced to learning Moore machines. Thus, we can apply off-the-shelf learning algorithms for Moore machines in order to learn QDAs. The resulting learning algorithms for QDAs enjoy many nice properties, such as that the number of queries and the runtime is bound polynomially in the size of the canonical QDA that is to be learned.

[2] The QDA in Figure 5.2 does not completely comply with the formal definition of QDAs. However, we skip details here for the sake of a more accessible description and refer the reader to Definition 5.4.

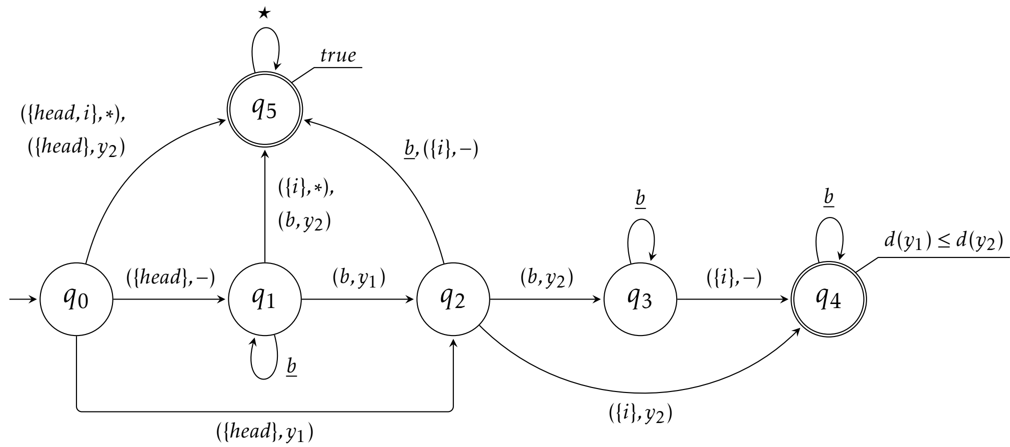


Figure 5.2: A QDA accepting the set of data words that correspond to program configurations satisfying Formula 5.1. Missing transitions point to a sink-state labeled with the formula *false*, which is not shown for the sake of readability. All states depicted as a single circle are implicitly labeled with the formula *false*. An "$*$" represents an arbitrary element of Y, whereas a "★" represents an arbitrary input-symbol.

As indicated above, a QDA is an alternative representation of a set of program configurations, namely those that correspond to a data word accepted by the QDA. Thus, our key idea is to learn a QDA and subsequently derive a logic formula (in Strand or the Array Property Fragment) from it that exactly characterizes the set of program configurations that the QDA represents. However, the class of logic formulas captured by QDAs is very expressive, and satisfiability of such formulas is undecidable in general. Consequently, even if we learn QDAs in an invariant learning application, it is impossible to verify automatically whether the learned properties are adequate invariants for the program at hand. The goal is, hence, to offer mechanisms to learn invariants that are amenable to decision procedures.

Elastic QDAs and a unique minimal over-approximation theorem In order to guarantee decidability of the resulting logic formulas, we exploit a common property of the Array Property Fragment and the decidable syntactic fragment of Strand called *elasticity* (following the general terminology in the literature on Strand [MPQ11, MQ11]). Intuitively, elasticity prohibits testing whether universally quantified variables reference cells that are a fixed distance away. The Array Property Fragment does this by disallowing arithmetic expressions over the quantified index variables, whereas the decidable syntactic fragment of Strand only permits the use of so-called *elastic relations* in order to relate universally quantified variables.

As an example, reconsider Formula 5.1 (on Page 135). This formula is *not* in the decidable syntactic fragment of Strand since the universally quantified variables y_1 and y_2 are related by the nonelastic successor relation $\rightarrow$ (in the subformula $y_1 \rightarrow y_2$). Similarly, Formula 5.2 is *not* in the Array Property Fragment because y_1 and y_2 are related by $y_2 = y_1 + 1$. A formula equivalent to Formula 5.1 but in the decidable syntactic fragment of Strand is

$$\forall y_1 : \forall y_2 : (head \rightarrow^* y_1 \land y_1 \rightarrow^* y_2 \land y_2 \rightarrow^* i) \rightarrow d(y_1) \leq d(y_2). \tag{5.3}$$

This formula compares the data of the cells referenced by y_1 and y_2 whenever y_2 occurs sometime after y_1 (as opposed to comparing the data of cells that are direct successors). This makes the formula fall into a decidable class.

Similarly, a formula equivalent to Formula 5.2 (on Page 136) but in the Array Property Fragment[3] is

$$\forall y_1 : \forall y_2 : (0 \leq y_1 \land y_1 \leq y_2 \land y_2 \leq i) \rightarrow a[y_1] \leq a[y_2]. \tag{5.4}$$

Again, replacing the condition that y_2 is the direct successor of y_1 with $y_1 \leq y_2$ makes the formula fall into a decidable class.

Based on the notion of elasticity, we identify a *structural restriction* of QDAs that permits only elastic properties to be expressed. This restriction allows us to define a subclass of QDAs, called *elastic QDAs (EQDAs)*, which we introduce in Section 5.3. In addition, we show two important results for EQDAs:

- A *unique minimal over-approximation theorem* stating that for every QDA, say accepting a language L_{val} of valuation words, there exists a *minimal* (with respect to inclusion) language of valuation words $L'_{val} \supseteq L_{val}$ that is recognizable by an EQDA.
- One can convert elastic QDAs into formulas of decidable logics: into the Array Property Fragment when modeling arrays and into the decidable syntactic fragment of Strand when modeling lists. We describe this translation in Section 5.4.

By combining both results, it is now possible to learn universally quantified properties that are amenable to automatic decision procedures: we first learn a QDA, then apply the unique minimal over-approximation (which is effective) to obtain the best over-approximation that can be expressed by elastic QDAs, and finally derive a formula in the Array Property Fragment, respectively the decidable fragment of Strand, from the resulting EQDA.

[3] To ease notation, we use the same relation symbols, such as $\leq$ and $=$, for both the element and the index logic throughout this chapter. Additionally, for the sake of a unified notation, we denote an array read operation at index i also by $d(i)$—instead of $a[i]$—if the array is clear from the context.

Passive learning of universally quantified invariants Our active learning algorithm for QDAs itself can be used in an invariant synthesis framework in which an (ICE-)teacher answers membership and equivalence queries about the program in question. One possible way to build a teacher is to use bounded and reverse-bounded Model Checking to answer membership queries and a logic solver to check whether a QDA provided on an equivalence query represents an adequate invariant. However, actually implementing such a teacher takes a lot of time and effort (if it is possible at all), and often requires specific adaptations (e.g., due to a potentially complex program logic and different data domains).

Therefore, we do neither pursue an active learning nor an (iterative) ICE-learning approach here.[4] Instead, we develop a passive learning algorithm (without the minimality constraint of the classical passive learning setting) that builds upon our active learning algorithm for QDAs. We believe—and we validate this belief by means of experiments—that a lighter-weight passive learning algorithm, which learns from a few randomly chosen small data structures, is in many situations sufficient to identify an invariant; by doing so, we rely on the observation that invariants are usually independent of the actual size and data of data structures.

The passive learning algorithm, which we present in Section 5.5, works as follows: First, we extract a finite set of program configurations by sampling the program in question (perhaps by just running the program on various small random inputs) and turn this set into a sample $\mathcal{S}$ of formula words. Second, we pit our active learning algorithm for QDAs against a teacher who answers queries with respect to $\mathcal{S}$ and makes sure that the QDA finally learned at least classifies $\mathcal{S}$ correctly. Once the learning has finished, one can translate the learned QDA into a logic formula and check (e.g., using a logic solver) whether it is an invariant for the program at hand.

However, since our teacher relies on a finite set of program configurations, he is necessarily imprecise and might err on queries (though he makes sure that any QDA passing an equivalence query at least agrees with the observed program behavior). This inaccuracy can prevent the learner from ever learning an invariant but is common in most learning algorithms employed in verification (e.g., in learning Boolean functions [KJD+10] as well as compositional verification [CGP03, AMN05a]). Nonetheless, in our experiments, which we discuss in Section 5.6, all QDAs that we learned represented an invariant for the program in question. This demonstrates that our combination of active and passive learning is a worthwhile approach to synthesize invariants.

Finally, we conclude this chapter in Section 5.7 with a summary and an outlook on future work.

[4] We recently presented an iterative ICE-learning algorithm for EQDAs [GLMN13a, GLMN14]. We comment on this result in Section 5.7.

The results of this chapter are joint work with Pranav Garg, Christof Löding, and P. Madhusudan, partly published in conference proceedings [GLMN13b] and as a technical report [GLMN13c].

Related Work

Shape analysis techniques [SRW02] are perhaps the best-known techniques to identify invariants expressing properties on dynamic heaps. In this context, heap locations are classified (merged) using unary predicates—some dictated by the program and some given as instrumentation predicates by the user—, and abstractions summarize all locations with the same predicates into a single one. The data automata that we build also express infinite sets of linear data structures (in that they accept words corresponding to data structures) and further allow n-ary quantified relations between elements of the data domain. In recent work, Bouajjani et al. [BDES12] described an abstract domain for analyzing list manipulating programs that can capture quantified properties about the structure and the data stored in lists. This domain can be instantiated with any numerical domain for the data constraints and a set of user-provided patterns for capturing the structural constraints. However, providing patterns for quantified invariants is a difficult task in general.

In recent years, techniques based on Craig's interpolation [McM03] have emerged as new methods for invariant synthesis. Interpolation techniques, which are inherently white-box techniques, are known for several theories, including linear arithmetic, uninterpreted function theories, and even for quantified properties over arrays and lists [JM07, McM08, ABG+12, SPW09]. These methods use different heuristics such as term abstraction [ABG+12], preferring smaller constants [JM07, McM08], or the use of existential ghost variables [SPW09] to ensure that the interpolant converges to an invariant starting from a finite set of spurious counterexamples. Moreover, IC3 [Bra11] is another white-box technique for generalizing inductive invariants from a set of counterexamples.

A primary difference in our work, compared to all the work above, is that ours is a black-box technique that does not look at the code of the program; instead, it synthesizes an invariant from a snapshot of examples and counterexamples that characterize the invariant. A black-box approach has both advantages and disadvantages. The main disadvantage is that information regarding what the program actually does is lost in invariant synthesis. However, this is the basis for its advantage as well—by *not* looking at the code, the learning algorithm promises to learn the sets with the simplest representations in polynomial time, and can also be much more flexible; for instance, even if the code of the program is complex (having nonlinear arithmetic or complex

heap manipulations that preclude logical reasoning), black-box learning can be used to learn simple invariants.

There exist several black-box learning algorithms that have been explored in verification; for instance, learning of Boolean formulas has been investigated for finding quantifier-free program invariants [CW12], which has subsequently been extended to quantified invariants by Kong et al. [KJD+10]. In contrast to our approach, however, the approach of Kong et al. requires a set of data predicates as well as the predicates allowed to appear in the guards of the quantified formula to be given.

Recently, *machine learning* techniques have also been explored [SNA12]. Variants of Houdini [FL01] essentially use conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas (also see Srivastava and Gulwani [SG09]). The most mature work in this area is Daikon [ECGN00], which learns formulas over a template by enumerating formulas and checking which ones satisfy the samples. Scalability is achieved in practice using several heuristics that reduce the enumeration space, which is doubly-exponential. For quantified invariants over data structures, however, such heuristics are not very effective, and Daikon often restricts learning to formulas of very restricted syntax such as formulas with a single atomic guard. In our experiments, for instance, Daikon was not able to learn an adequate loop invariant for the selection sort example.

5.1 Quantified Data Automata

Before we introduce QDAs, we first need to define on which kind of input these automata operate. To this end, we introduce three types of words: *data words*, *valuation words*, and *symbolic words*.

We model lists and arrays (and also finite sets of lists and arrays) that contain data over some (infinite) data domain D as data words. Intuitively, data words encode the structure of the array or list, the data values stored in the cells, and also the (finite) set of pointer variables used by the given program. Thereby, each symbol of a data word corresponds to a cell of the data structure in the order of their occurrence. To define data words formally, we fix a finite (potentially empty) set of pointer variables $PV = \{p_1, \ldots, p_r\}$.

Definition 5.1 (Data word)**.** Let PV be a finite set of pointer variables, $\Sigma = 2^{PV}$, and D a (potentially infinite) data domain (i.e., a set of data values). A *data word* over PV and D is a word $u \in (\Sigma \times D)^*$ where each $p \in PV$ occurs exactly once in the first component of u (i.e., for each $u = a_1 \ldots a_n$ and $p \in PV$, there exists precisely one $j \in \{1, \ldots, n\}$ such that $a_j = (X, d)$ and $p \in X$).

The empty set in the first component of a data word corresponds to a blank-symbol, which we denote by the symbol b. Such a blank-symbol indicates that no pointer variable points to the corresponding cell.

Let us fix a finite, nonempty set $Y = \{y_1, \dots, y_k\}$ of *universally quantified variables*. Intuitively, the automata we build do not read data words directly but valuation words, which make the universal quantification explicit. More precisely, a valuation word is a data word extended by an additional component, called a *valuation of Y*, that encodes the cells the variables from Y reference (similar to the Σ-component of data words). The variables from Y are then quantified universally in the semantics of the automaton model (as explained later in this section).

Definition 5.2 (Valuation word)**.** A *valuation word* is a word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ where v projected to the first and third components forms a data word and where each $y \in Y$ occurs in the second component of v precisely once.

We use the symbol "$-$" for positions at which no variables of Y occur. Note that the choice of the alphabet enforces the variables of Y to be in different positions.

A valuation word defines a data word along with a valuation of Y. The data word corresponding to a valuation word v is the word $\mathrm{dw}(v) \in (\Sigma \times D)^*$ obtained by projecting v to its first and third components.

In later parts of this chapter, we use a third type of words, which we call *symbolic words*. In contrast to data and valuation words, symbolic words capture the structure of a list or array but do not contain data.

Definition 5.3 (Symbolic word)**.** Let $\Sigma = 2^{PV}$ and $\Pi = \Sigma \times (Y \cup \{-\})$. A *symbolic word* is a word $w \in \Pi^*$ where each $p \in PV$ occurs precisely once in the first component of w and each $y \in Y$ occurs precisely once in the second component of w.

We denote the symbol in Π representing that neither a pointer variable nor a universally quantified variable occurs by $\underline{b} = (b, -)$. Analogous to valuation words, the symbolic word corresponding to a valuation word v is the word $\mathrm{sw}(v) \in \Pi^*$ obtained by projecting v to its first two components.

Example 5.1. Consider the valuation word shown in Figure 5.3 (on Page 144). Besides the valuation word itself, Figure 5.3 illustrates which components of the valuation word constitute the corresponding data word and which the corresponding symbolic word. Note that Figure 5.3 depicts symbols of the word as column-vectors for the sake of readability, whereas we usually use row-vectors in depictions of QDAs (e.g., as in Figure 5.1 on Page 137). ◁

To express properties on the data, we fix a set of constants, functions, and relations over the data domain. We assume that the quantifier-free first-order theory over this

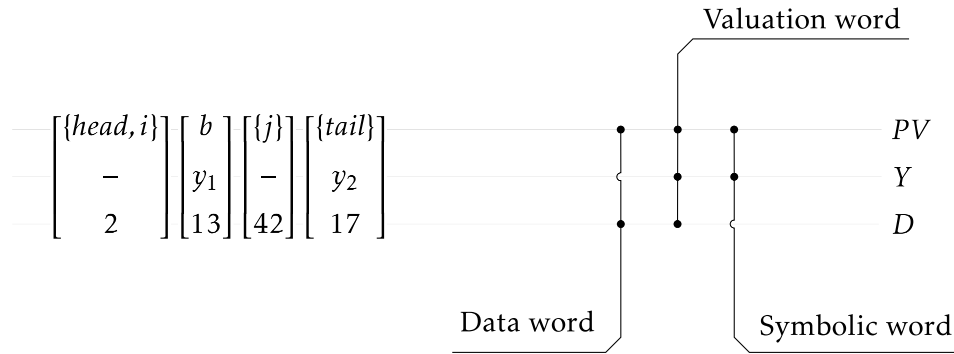


Figure 5.3: A valuation word together with a depiction of which components constitute the corresponding data word and symbolic word.

domain is decidable. We encourage the reader to keep in mind the theory of integers with constants 0, 1, etc., addition, and the usual relations $\leq$, $<$, $=$, etc. as a standard example of such a domain.

A quantified data automaton uses a finite set F of *data formulas* over the atoms $d(y_1),\dots,d(y_k)$ that refer to the data values at the cells referenced by the variables $y_1,\dots,y_k$. Moreover, we assume that F forms a (semi-)lattice $\mathcal{F} = (F, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ where $\sqsubseteq$ is the partial-order relation over F, $\sqcup$ is the least-upper bound, and *false* and *true* are formulas required to be in F that correspond to the bottom and top elements of the lattice, respectively. Furthermore, we assume that whenever $\alpha \sqsubseteq \beta$, then $\alpha \rightarrow \beta$. Finally, we require formulas in the lattice to be pairwise *inequivalent*.

One obtains an example of such a formula lattice over the data domain of integers by taking a set of representatives of all inequivalent Boolean formulas over the atoms involving no constants, defining $\alpha \sqsubseteq \beta$ if and only if $\alpha \rightarrow \beta$, and taking the least-upper bound of two formulas as their disjunction. Such a lattice is of size doubly exponential in the number of variables, and, consequently, unsuitable in practice. Thus, one might want to use a different, coarser lattice, such as the Cartesian lattice.

The *Cartesian lattice* is formed over a set of atomic formulas and consists of conjunctions of literals (atoms or negations of atoms). The least-upper bound of two formulas is the conjunction of those literals that occur in both formulas; for example, if the set of atomic formulas is $\{\varphi_1,\dots,\varphi_4\}$, $\alpha = \varphi_1 \wedge \neg\varphi_2 \wedge \varphi_3$, and $\beta = \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, then $\alpha \sqcup \beta = \varphi_3$ because this is the only literal that occurs in both α and β. For the ordering, we define $\alpha \sqsubseteq \beta$ if and only if all literals appearing in β also appear in α. Note that the size of a Cartesian lattice is only exponential in the number of literals.

We have now introduced all necessary concepts and are ready to define the automaton model.

Definition 5.4 (Quantified data automaton)**.** Let PV be a finite set of pointer variables, D a data domain, Y a finite, nonempty set of universally quantified variables, and $\mathcal{F}$ a formula lattice over a finite set F of formulas. A *quantified data automaton (QDA)* is a tuple $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ where Q is a finite, nonempty set of states, $\Pi = \Sigma \times (Y \cup \{-\})$ is the input alphabet, $q_0 \in Q$ is the initial state, $\delta\colon Q \times \Pi \dashrightarrow Q$ is the (partial) transition function, and $f\colon Q \to F$ is the *final-evaluation function*, which maps each state to a data formula.

Intuitively, a QDA is a register automaton that is equipped with a register for each universally quantified variable $y \in Y$. A QDA reads a valuation word, stores the data located at the positions referenced by the variables in Y, and checks whether the formula decorating the state finally reached holds for the data in the registers. It accepts a data word $u \in (\Sigma \times D)^*$ if it accepts *all possible* valuation words that extend u with a valuation of Y.

Before we define the semantics of QDAs formally, let us briefly comment on why we allow QDAs to access data solely at cells referenced by universally quantified variables. There are two reasons for this decision: First, granting access also to the data at other cells (particularly referenced by pointer variables) introduces subtle but serious problems, which we want to avoid. Second, most (natural) invariants do not express properties of the data at cells referenced by pointer variables; due to the unbounded nature of arrays and lists, invariants typically state conditions that need to be satisfied by all—or arbitrary—cells and can be expressed solely by relating the data at cells referenced by universally quantified variables. In addition, the formula lattice is smaller (as it contains less atoms), which is advantageous in the context of learning QDAs. Though the limited access to the data restricts the invariants expressible by QDAs, our experiments show that this is not a concern in applications.

Let us now formalize the semantics of QDAs. Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, a *configuration* of $\mathcal{A}$ is a pair (q, r) where $q \in Q$ is a state and $r\colon Y \dashrightarrow D$ is a *partial variable assignment* that assigns a value of the data domain to a universally quantified variable. The initial configuration is (q_0, r_0) where the domain of r_0 is empty.

The *run* of $\mathcal{A}$ on a valuation word $v = (a_1, y_1, d_1) \dots (a_n, y_n, d_n) \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ is a sequence $(q_0, r_0), \dots, (q_n, r_n)$ of configurations that satisfies $\delta(q_i, (a_i, y_i)) = q_{i+1}$ and

$$r_{i+1} = \begin{cases} r_i\{y_i \leftarrow d_i\} & \text{if } y_i \in Y; \\ r_i & \text{if } y_i = -; \end{cases}$$

for all $i \in [n]$ (recall that $r_i\{y_i \leftarrow d_i\}$ corresponds to the mapping r_i in which y_i is mapped to d_i). As in the case of DFAs, we use $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q_n, r_n)$ as a shorthand-notation.

The QDA $\mathcal{A}$ *accepts* a valuation word v if $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q, r)$ where (q_0, r_0) is the initial configuration and $r \models f(q)$; that is, after reading the valuation word, the data stored in the registers satisfies the formula annotating the state finally reached. The language $L_{val}(\mathcal{A})$ is the set of valuation words accepted by $\mathcal{A}$.

The QDA $\mathcal{A}$ *accepts* a data word $u \in (\Sigma \times D)^*$ if $\mathcal{A}$ accepts all valuation words v with $\mathrm{dw}(v) = u$. The language $L_{dat}(\mathcal{A})$ is the set of data words accepted by $\mathcal{A}$.

To ease working with QDAs and to obtain the intended semantics, we assume throughout this chapter that each QDA satisfies two further constraints:

- Each QDA verifies that its input satisfies the constraints on the number of occurrences of variables from PV and Y. All inputs violating these constraints (i.e., all inputs that are not valuation words) either do not admit a run due to missing transitions or lead to a dedicated state labeled with the data formula *false*. This property implies that the states of an QDA are "typed" with the set of variables that have been read so far. As a consequence, cycles in the transition structure of an QDA can only be labeled with $\underline{b}$-symbols. Note that this assumption is no restriction because both the language of valuation words and the language of data words are defined in terms of words that satisfy the correct occurrence of variables from PV and Y.

- Each QDA verifies that the universally quantified variables occur in its input in the same fixed order, say $y_1 \prec \cdots \prec y_k$. All valuation words violating this order lead to a dedicated state labeled with the data formula *true* (i.e., all such valuation words are accepted). The rationale behind this assumption is the following: since the variables $y \in Y$ are universally quantified, it is sufficient to check a property with respect to a fixed order and a different order should not change the accepted language of data words.

  Although this assumption is a restriction in general, each QDA can be transformed into one that accepts the same data language and respects the predetermined variable ordering if the formula lattice is closed under conjunction. The idea for such a construction is to use a subset construction that follows all paths that only differ in the order of Y. For each state in a set of states reached like that, one remembers in which order the variables in Y have occurred. At the final states, one uses the conjunction of all formulas in the set with the appropriate renaming of the variables in Y. Due to the universal semantics of QDAs, this results in a QDA that accepts the same data language as original automaton. Since most natural formula lattices, such as the full lattice and the Cartesian lattice (which we use in this chapter), are closed under conjunction, we can without loss of generality assume that each QDA respects a fixed ordering of the universally quantified variables.

5.2 Learning Quantified Data Automata

Our goal in this section is to learn QDAs using existing active learning algorithms, such as Angluin's algorithm, which was developed to infer the canonical DFA for a regular language. Therefore, we begin this section by analyzing the notion of canonical automata for QDAs. The result of this analysis then allows us to reduce the learning of QDAs to the learning of Moore machines.

5.2.1 Canonical QDAs

Recall that QDAs define two kinds of languages, namely a data word language and a valuation word language. We begin by observing that we cannot hope for unique minimal QDA on the level of data words.

To see why, consider the QDA $\mathcal{A}$ depicted in Figure 5.4 (on Page 148) over $PV = \emptyset$ and $Y = \{y_1, y_2\}$. It accepts all valuation words in which

- $d(y_1) \leq d(y_2)$ if y_1 occurs before y_2 and y_1, y_2 are both on even positions;
- $y_2 < y_1$; or
- at least one of y_1 and y_2 does not occur at an even position.

Hence, $\mathcal{A}$ accepts the data word language that consist of all data words for which the data at even positions is sorted. Since each QDA has to ensure that each variable occurs exactly once, the number of states of $\mathcal{A}$ is minimal for defining this language of data words.

However, a QDA in which we replace the transition $\delta(q_6, \underline{b}) = q_5$ by the transition $\delta(q_6, \underline{b}) = q_1$ accepts the same language of data words. This new QDA checks the sortedness only for all y_1, y_2 with $y_2 = y_1 + 2$, which is sufficient. This shows that the transition structure of a state-minimal QDA for a given language of data words is not unique.

On the level of valuation words, on the other hand, we can show the existence of canonical minimal automata.

Theorem 5.1. *For each QDA $\mathcal{A}$ there is a unique minimal QDA $\mathcal{A}_{min}$ that accepts the same set of valuation words.*

An intuitive explanation is that the automaton model is deterministic and since all universally quantified variables are in different positions, a QDA cannot derive any information about the relation between the data during its run. Let us prove Theorem 5.1 formally.

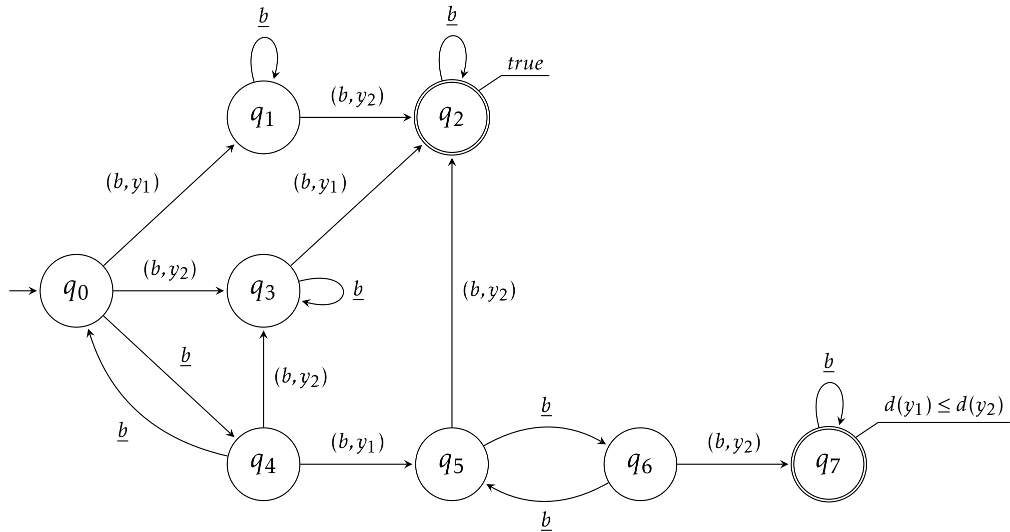


Figure 5.4: A QDA expressing the property over lists that the data on even positions is sorted. Missing transitions lead to a sink-state labeled with *false*, which is not shown for the sake of readability. All states drawn as single circle are implicitly labeled with the formula *false*.

Proof of Theorem 5.1. Consider a language $L_{val} \subseteq (\Pi \times D)^*$ of valuation words that can be accepted by a QDA, and let $w \in \Pi^*$ be a symbolic word. Then, there has to be a formula φ_w in the lattice that precisely characterizes all valuation words $v \in L_{val}$ that extend w with data (i.e., that satisfy $\mathrm{sw}(v) = w$). Since we assume all formulas in the lattice to be pairwise nonequivalent, φ_w is uniquely determined. One obtains φ_w by considering for each valuation word v with $\mathrm{sw}(v) = w$ the greatest lower bound φ_v of all formulas in the lattice that v satisfies and then taking the least upper bound of all these φ_v.

In fact, the formula φ_w is independent of a particular QDA accepting L_{val}. To see why, consider two QDAs, say $\mathcal{A}$ and $\mathcal{A}'$, that both accept L_{val}. In addition, assume that the QDA $\mathcal{A}$ reaches state q on reading w and that the QDA $\mathcal{A}'$ reaches state q' on reading w. In this situation, both q and q' have to be labeled with the same data formula because $\mathcal{A}$ and $\mathcal{A}'$ would otherwise accept different languages of valuation words. This proves that φ_w is unique and only depends on the given language of valuation words.

Thus, a language of valuation words can be seen as a function that assigns a formula to every symbolic word, and one can think of a QDA as a Moore machine that computes this function. Moreover, for each Moore machine, there exists a unique minimal Moore machine that computes the same function (e.g., see Kohavi [Koh70]). This proves Theorem 5.1. □

5.2.2 Learning QDAs by Learning Moore Machines

The proof of Theorem 5.1 suggests viewing QDAs as Moore machines, and our goal is to use existing learning algorithms for Moore machines to learn QDAs. To this end, we separate the structure of valuation words (i.e., the length of the words, the cells the pointer variables point to, and so on) from the data contained in the words. We do so by introducing what we call *formula words*.

Definition 5.5 (Formula word)**.** Let PV be a finite set of pointer variables, Y a finite, nonempty set of universally quantified variables, and $\mathcal{F}$ a lattice over a finite set F of formulas. A *formula word* is a pair $(w, \varphi) \in (\Pi^* \times F)$ consisting of a symbolic word $w \in \Pi^*$ (as before, $\Sigma = 2^{PV}$ and $\Pi = \Sigma \times (Y \cup \{-\})$) and a data formula $\varphi \in F$.

Note that a formula word does not contain elements of the data domain—it simply consists of the symbolic word that depicts the pointers into the list (modeled using Σ), a valuation for the quantified variables (modeled using $Y \cup \{-\}$), as well as a formula over the data domain. Hence, a symbolic word represents a set of valuation words, namely those whose data component satisfies the data formula.

Example 5.2. The formula word

$$\Big((\{head\}, y_1)\underline{b}(b, y_2)(\{tail\}, -),\ d(y_1) \leq d(y_2)\Big)$$

represents that *head* and y_1 point to the first cell of a list, y_2 points to the third cell, and *tail* points to the last cell; the data formula is $d(y_1) \leq d(y_2)$. ◃

A QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ over the set F of data formulas accepts a formula word $(w, \varphi) \in \Pi^* \times F$ if $\mathcal{A}$ reaches a state $q \in Q$ on reading the symbolic word w and $f(q) = \varphi$. Given a QDA $\mathcal{A}$, we define the language $L_{for}(\mathcal{A}) \subseteq \Pi^* \times F$ of formula words accepted by $\mathcal{A}$ in the usual way. Moreover, we call a language $L \subseteq \Pi^* \times F$ of formula words *QDA-acceptable* if there exists a QDA $\mathcal{A}$ with $L_{for}(\mathcal{A}) = L$.

Note that not every language of formula words is QDA-acceptable; for instance, consider the language

$$L^\star_{for} = \{(\underline{b}^i(p, y)\underline{b}^i,\ true) \mid i \geq 1\}.$$

A standard pumping argument shows that $L^\star_{for}$ cannot be accepted by a QDA since the number of blanks at the beginning and at the end of a word have to match. Furthermore, words whose symbolic component is not of the form $\underline{b}^i(p, y)\underline{b}^i$ are not present in $L^\star_{for}$ but a QDA necessarily assigns a unique formula to every symbolic word.

In fact, every QDA-acceptable language L_{for} of formula words has to fulfill at least the following three constraints:

- For every symbolic word $w \in \Pi^*$, there exists a formula φ such that $(w, \varphi) \in L_{for}$.
- If $(w, \varphi) \in L_{for}$ and $(w, \varphi') \in L_{for}$, then $\varphi = \varphi'$.
- The number of different formulas occurring in formula words in L_{for} is finite.

These constraints allow us to treat QDAs as Moore machines that read symbolic words and output data formulas. In fact, a QDA-acceptable language $L_{for} \subseteq \Pi^* \times F$ is an alternative representation of a Moore machine-computable mapping $f: \Pi^* \to F$. One easily deduces that two two QDAs (over the same lattice of formulas) that accept the same set of valuation words also accept the same set of formula words (assuming that all formulas in the lattice are pairwise nonequivalent). Thus, we can easily reduce the problem of learning QDAs to the problem of learning Moore machines. Note that we intentionally do not view a QDA as a device that computes a mapping but as a device that accepts a language. We do this to ease the description in later sections.

Before we describe how to reduce the learning of QDAs to the learning of Moore machines, let us briefly discuss the latter.

Actively Learning Moore Machines

In the context of actively learning Moore machines, the target concept is a Moore machine computable function $f: \Sigma^* \to \Gamma$ that maps each word u over the input alphabet Σ to an output $f(u)$ taken from an output alphabet Γ. Note that we obtain Angluin's original setting by letting $\Gamma = \{0, 1\}$.

Given a Moore machine-computable function $f: \Sigma^* \to \Gamma$, a teacher for f answers queries as follows.

Membership query On a membership query with a word $u \in \Sigma^*$, the teacher returns the function value $f(u)$.

Equivalence query On an equivalence query with a Moore machine $\mathcal{M}$, the teacher checks whether $f_{\mathcal{M}} = f$ is satisfied. If this is the case, he returns "yes". If this is not the case, he returns a counterexample $u \in \Sigma^*$ with $f(u) \neq f_{\mathcal{M}}(u)$.

Note that the learner and teacher do not need to agree a priori on the output alphabet since the learner can obtain this knowledge through membership queries.

One can straightforwardly adapt observation table-based learning algorithms, such as Angluin's algorithm and Rivest and Schapire's algorithm, to learn Moore machines. The idea is to lift the Nerode congruence to Moore machine-computable functions $f: \Sigma^* \to \Gamma$ by defining

$$u \sim_f v \text{ if and only if } \forall w \in \Sigma^*\colon f(uw) = f(vw),$$

where $u, v \in \Sigma^*$. Then, it is indeed enough to adapt the mapping T of an observation table and the way conjectures are generated.

More precisely, one changes the mapping T of an observation table $O = (R, S, T)$ to a mapping $T\colon (R \cup R \cdot \Sigma) \cdot S \to \Gamma$. Moreover, one does no longer produce a DFA as conjectures but a Moore machine $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta, \lambda)$ whose output function is defined by $\lambda([\![u]\!]_O) = T(u)$ where $u \in R$. Everything else (i.e., the notion of O-equivalence, the notion of closedness and consistency, and the functioning of the algorithm) is left unchanged. Chen et al. [CFC+09] demonstrate this adaptation for the case $|\Gamma| = 3$.

These so adapted algorithms learn the unique minimal Moore machine for the target function in time polynomial in the size of the minimal Moore machine and the length of the longest counterexample returned by the teacher. Thus, we immediately obtain the following remark.

Remark 5.1. Given a teacher for a Moore machine-computable function, who can answer membership and equivalence queries, the unique minimal Moore machine for this function can be learned in time polynomial in the size of the minimal Moore machine and the length of the longest counterexample returned by the teacher.

Actively Learning QDAs

For the task of actively learning QDAs, we assume that the teacher has access to a QDA-acceptable language $L \subseteq \Pi^* \times F$ of formula words and answers queries as follows.

Membership query On a membership query, the learner provides a symbolic word $w \in \Pi^*$, and the teacher returns the unique formula $\varphi \in F$ with $(w, \varphi) \in L$. Note that such a formula word is guaranteed to exist since L is a QDA-acceptable language.

Equivalence query On an equivalence query with a QDA $\mathcal{A}$, the teacher checks whether $L_{for}(\mathcal{A}) = L$ is satisfied. If this is the case, he returns "yes". If this is not the case, then there exists a formula word (w, φ) such that $(w, \varphi) \in L_{for}(\mathcal{A})$ if and only if $(w, \varphi) \notin L$ (since both $L_{for}(\mathcal{A})$ and L contain a formula word of the form (w', φ') for every $w' \in \Pi^*$), and the teacher returns w as counterexample.

Since a teacher for QDAs answers queries in the same manner as a teacher for Moore machines and each QDA-acceptable language contains only finite many different data formulas, we can reduce the learning of QDAs to the learning of Moore machines. This allows us to apply off-the-shelf learning algorithms, such as Angluin's or Rivest and Schapire's algorithm, and we obtain the following result.

Theorem 5.2. *Given a teacher for a QDA-acceptable language of formula words, who can answer membership and equivalence queries, the unique minimal QDA for this language*

can be learned in time polynomial in the size of the minimal QDA and the length of the longest counterexample returned by the teacher.

We end this section by remarking that learning QDAs on the level of valuation words (or formula words) has the drawback that one has to fix a set of valuation words that represents the language of data words one is interested in. In Section 5.5, we present an implementation of a teacher who answers queries based on information derived from runs of actual code manipulating lists and arrays. Such a teacher does not know an invariant and is necessarily imprecise. Thus, the teacher might answer the queries with respect to a language of valuation words that requires a large QDA or is not even QDA-acceptable at all. However, in the setting of learning invariants, a learner does not need to learn the exact language the teacher "has in mind". It suffices if a learner arrives at some QDA that represents an invariant. Since invariants are often not very complex, the hope is that a learner succeeds in learning an invariant from the incomplete information provided by the teacher. We substantiate this hope in Section 5.6, where we present results on learning invariants for various programs.

5.3 Elastic Quantified Data Automata

Our aim is to translate the QDAs that we learned into decidable logics such as the decidable syntactic fragment of Strand or the Array Property Fragment. A property shared by both logics is that they cannot test whether two universally quantified variables reference cells that are a fixed distance away. We capture this type of constraint by the subclass of *elastic QDAs*.

Definition 5.6 (Elastic QDA). A QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ is called *elastic* if each transition on $\underline{b}$ is a self-loop (i.e., whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$).

The intuition behind this definition is to disallow a QDA to relate two universally quantified variables via a bounded number of $\underline{b}$-transitions; a self-loop on $\underline{b}$, on the other hand, cannot bound the distance between variables and, hence, corresponds to an elastic relation. It might seem that missing $\underline{b}$-transitions enable EQDAs to test whether two universally quantified variables are a bounded distance away. However, due to the universal semantics of the automaton model, such a test is not possible in general but only if the variables can be related to a pointer variable. We discuss this in more detail in the translation from EQDAs to logic formulas in Section 5.4.2, where we introduce the notion of irrelevant self-loop.

The learning algorithm that we use to synthesize QDAs does not construct EQDAs in general. However, we can show that every QDA can be *uniquely over-approximated* by a language of valuation words that can be accepted by an EQDA. We refer to this

construction, which we define below, as *elastification*. Elastification crucially relies on the particular structure of elastic QDAs, which forces a unique set of valuation words to be added to the original language in order to make it elastic.

To ease the following definition, let us introduce a few auxiliary notations. Given a QDA $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$, let $R_{\underline{b}}(q) \subseteq Q$ be the set of states reachable from q via a (possibly empty) sequence of $\underline{b}$-transitions and $R_{\underline{b}}(S) = \bigcup_{q \in S} R_{\underline{b}}(q)$ for a set $S \subseteq Q$. Moreover, we lift the transition function δ to sets in the usual way: for $S \subseteq Q$ and $a \in \Pi$, let $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$.

Definition 5.7 (Elastification)**.** Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, we define the EQDA $\mathcal{A}_{el} = (Q_{el}, \Pi, q_0^{el}, \delta_{el}, f_{el})$ by

- $Q_{el} = \{S \mid S \subseteq Q\}$;
- $q_0^{el} = R_{\underline{b}}(q_0)$;
- $f_{el}(S) = \bigsqcup_{q \in S} f(q)$ where $S \subseteq Q$; and
- $\delta_{el}(S, a) = \begin{cases} R_{\underline{b}}(\delta(S, a)) & \text{if } a \neq \underline{b}; \\ S & \text{if } a = \underline{b} \text{ and } \delta(q, \underline{b}) \text{ is defined for a } q \in S; \\ \text{undefined} & \text{otherwise.} \end{cases}$

Note that the construction of Definition 5.7 is similar to the usual powerset construction, except that we take the "$\underline{b}$-closure" after applying the transition function of $\mathcal{A}$. Moreover, $\mathcal{A}_{el}$ loops in a state S if a $\underline{b}$-transitions is defined for a state $q \in S$.

Let us now show that $L_{val}(\mathcal{A}_{el})$ is the most precise elastic over-approximation of $L_{val}(\mathcal{A})$.

Theorem 5.3. *Let $\mathcal{A}$ be a QDA and $\mathcal{A}_{el}$ the EQDA constructed according to Definition 5.7. Then, the following holds:*

- $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{el})$.
- *for every EQDA $\mathcal{B}$ with $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, the inclusion $L_{val}(\mathcal{A}_{el}) \subseteq L_{val}(\mathcal{B})$ holds.*

Proof of Theorem 5.3. We first observe that $\mathcal{A}_{el}$ is elastic by definition of δ_{el}. Moreover, a standard induction over the length of valuation words $v = a_1 \dots a_n \in (\Pi \times D)^*$ shows that if the run of $\mathcal{A}$ on v is

$$\mathcal{A}\colon q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n,$$

then the run of $\mathcal{A}_{el}$ on v is

$$\mathcal{A}_{el}\colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n$$

with $q_i \in S_i$ for all $i \in \{0,\dots,n\}$. Thus, $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{el})$ because the implication $f(q_n) \rightarrow f_{el}(S_n)$ holds by definition of f_{el} (and due to the fact that the set of data formulas forms a lattice). This proves the first part of Theorem 5.3.

Let us now show the second part of Theorem 5.3, namely that $L_{val}(\mathcal{A}_{el})$ is the most precise elastic over-approximation of $L_{val}(\mathcal{A})$. To this end, let $\mathcal{B} = (Q_{\mathcal{B}}, \Pi, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, f_{\mathcal{B}})$ be an EQDA with $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. In addition, let $v \in L_{val}(\mathcal{A}_{el})$. Thus, the task is to prove that $v \in L_{val}(\mathcal{B})$ also holds.

Assume S to be the state reached by $\mathcal{A}_{el}$ on reading v and p the state reached by $\mathcal{B}$ on reading v. We now show that $f(q)$ implies $f_{\mathcal{B}}(p)$ for each $q \in S$. Once we have established this, we obtain that $f_{el}(S)$ implies $f_{\mathcal{B}}(p)$ because $f_{el}(S)$ is the least formula in the formula lattice that is implied by all formulas $f(q)$ where $q \in S$. In addition, since $v \in L_{val}(\mathcal{A}_{el})$, the valuation word v satisfies $f_{el}(S)$ and, hence, also $f_{\mathcal{B}}(p)$. Thus, $v \in L_{val}(\mathcal{B})$.

To prove that $f(q)$ implies $f_{\mathcal{B}}(p)$ for each $q \in S$, pick a state $q \in S$. Following the definition of δ_{el}, we construct a valuation word $v' \in (\Pi \times D)^*$ that satisfies the following three properties:

- The QDA $\mathcal{A}$ accepts v'.
- The run of $\mathcal{A}$ on v' leads to state q.
- The run of $\mathcal{B}$ on v' leads to state p.

We obtain v' by amending v with symbols of the form $(\underline{b}, d)$ where $d \in D$ is an arbitrary value from the data domain. Since the data value at such positions does not occur in conjunction with variables, their value is unimportant.

For the actual construction of v', assume $v = a_1 \dots a_n$, let

$$\mathcal{A}_{el}\colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n$$

be the run of $\mathcal{A}_{el}$ on v (thus, $S = S_n$), and let $q \in S$. Based on this run, we derive states $q_i, q_i' \in S_i$ as well as natural numbers k_i where $i \in [n+1]$ such that

$$\mathcal{A}\colon q_0 \xrightarrow{\underline{b}^{k_0}} q_0' \xrightarrow{a_1} q_1 \xrightarrow{\underline{b}^{k_1}} q_1' \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \xrightarrow{\underline{b}^{k_n}} q_n'$$

is a run of $\mathcal{A}$ (which we prove to be accepting shortly).

We begin this construction by setting $q_n' = q$. Since $\delta_{el}(S_{n-1}, a_n) = S_n$ and $q_n' \in S_n$, we can choose states $q_{n-1}' \in S_{n-1}$ and $q_n \in S_n$ such that $\delta(q_{n-1}', a_n) = q_n$ and $q_n' \in R_{\underline{b}}(q_n)$, say $\mathcal{A}\colon q_n \xrightarrow{\underline{b}^{k_n}} q_n'$ for a suitable $k_n \geq 0$. We continue this construction as follows: given $q_j' \in S_j$ where $j \in \{1,\dots,n\}$, we choose q_{j-1}', q_j, and k_j as above; for $j = 0$, we pick k_0 such that $\mathcal{A}\colon q_0 \xrightarrow{\underline{b}^{k_0}} q_0'$. Finally, we set $v' = \underline{b}^{k_0} a_1 \underline{b}^{k_1} \dots a_n \underline{b}^{k_n}$.

By construction, the run of $\mathcal{A}$ on v' leads to state $q = q'_n$. Since v' is obtained from v by inserting $\underline{b}$-symbols, the valuation word v' also satisfies the formula $f(q)$ and, thus, $v' \in L_{val}(\mathcal{A})$. It remains to show that the run of $\mathcal{B}$ on v' leads to state p. Since $\mathcal{B}$ is elastic, the only possibility that v' does not lead to p in $\mathcal{B}$ is a missing self-loop on $\underline{b}$ at a position at which we inserted a nonempty sequence of $\underline{b}$-symbols. However, since $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, such a situation cannot exist.

Finally, we conclude that $f(q)$ implies $f_{\mathcal{B}}(p)$ using the following argument: If $f(q)$ does not imply $f_{\mathcal{B}}(p)$, then there exist data values at the position the variables $y_1, \ldots, y_k$ point to such that $f(q)$ is satisfied but $f_{\mathcal{B}}(p)$ is not. By changing the data values in v' accordingly, we can produce a valuation word that is accepted by $\mathcal{A}$ but not by $\mathcal{B}$. However, this contradicts the assumption $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. Thus, $f(q)$ implies $f_{\mathcal{B}}(p)$. □

5.4 Converting Program Configurations to Data Words and EQDAs to Logics

After having introduced elastic QDAs, we can now tie things together and describe how to convert program configurations to data words (in Section 5.4.1) and EQDAs to formulas in the Array Property Fragment, respectively in the syntactic decidable fragment of Strand (both in Section 5.4.2).

5.4.1 Modeling Program Configurations as Data Words

We model program configurations consisting of scalar variables, pointer or index variables,[5] and one (or more) linear data structures—lists or arrays in our case—as data words over a finite set of variables. The resulting data word is over the same domain D as the data in the cells of the data structure.

To simplify our modeling, we replace each scalar variable with an auxiliary pointer variable that points to a cell containing the data of the scalar variable. More precisely, for each scalar variable, we introduce a new pointer variable and extend the data structure with a new cell, which is located before the actual data structure begins and contains the data of the scalar variable; the order in which scalar variables are represented in the data structure is arbitrary but needs to be fixed. To be able to access the data at these positions (recall that QDAs can only access the data at position pointed to by universally quantified variables), we amend QDAs with a register for each such pointer variable and extend the set F of formulas over which the considered QDA works with the atom $d(x)$ for each scalar variable x.

[5] Index variables occur in the case of arrays and are used interchangeably with pointer variables.

Let c be a program configuration over a linear data structure and a finite set PV of pointer or index variables, and let $\Sigma = 2^{PV}$. Roughly speaking, we model c as the data word

$$u_c = (a_1, d_1) \dots (a_n, d_n) \in (\Sigma \times D)^*$$

such that the i-th symbol of the data word corresponds to the i-th cell of the data structure. In particular, the symbol $a_i \subseteq PV$ contains all pointer or index variables referencing the i-th cell, and d_i is the data stored in that cell. We encourage the reader to reconsider Figure 5.1 (on Page 137) and Example 5.3 below for a pictorial illustration of this encoding.

In the case of programs working over lists, some of the pointer variables might be *null* or point to unallocated memory, which cannot be dereferenced. We capture this situation in the data word by introducing an auxiliary pointer variable *nil* that points to a new cell at the beginning of the list. All pointer variables that are *null* or point to unallocated memory occur together with *nil*. The data value of the *nil* cell in the data word is not important and can be set to an arbitrary element of D.

Similarly, we introduce two new index variables *index_le_zero* and *index_geq_size* for arrays to capture index variables that are out-of-bounds (we assume that arrays are indexed starting at 0). The variable *index_le_zero* occurs together with all index variables that are less than zero, and *index_geq_size* occurs with those index variables that are either equal to or exceed the size of the array. Let the set Aux contain all auxiliary variables that may occur in our encoding.

To model configurations of programs that manipulate more than one data structure, one can use one of the following two approaches: the first approach concatenates the data structures using a special pointer variable $\star_i$ to demarcate the end of the i-th data structure; the second approach models several data structures as one single combined data structure over an extended data domain (e.g., consisting of several components analogous to the convolution of words, see Section 2.1). In our experiments, we followed the approach that suited the given situation best.

Let us illustrate our encoding of program configurations by an example.

Example 5.3. Consider a program that takes a scalar variable k ("key") and a list l as input and partitions l into two separate lists such that the first list contains all cells whose data value is less than k and the second list contains all the remaining cells. (This program occurs as *list-partition* in our experiments.) Let us assume that the program maintains the pointer variables h_1 ("head"), c_1 ("current"), and p to reference cells of the first list as well as h_2 and c_2 to reference cells of the second list.

The program works as follows: it scans l starting at h_1 step-by-step and whenever it encounters a data value equal to or greater than k, it moves this cell to the end of the

second list (as the new successor of the cell c_2 points to). Then, it updates the pointers accordingly. Thus, the data at cells from h_1 to p, which demarcates the end of the first list, is less than k, and the data at cells from h_2 to c_2 is greater than or equal to k.

Let us assume a concrete scenario where $k = 6$ and l is a list containing the data values 1 to 9 in ascending order. Consider the program configuration where the first seven cells of the list have been processed and c_1 is pointing to the cell with data value 8. A data word corresponding to this program configuration is

$$\begin{bmatrix}\{k\}\\6\end{bmatrix}\begin{bmatrix}\{nil\}_\end{bmatrix}\begin{bmatrix}\{h_1\}\\1\end{bmatrix}\begin{bmatrix}b\\2\end{bmatrix}\begin{bmatrix}b\\3\end{bmatrix}\begin{bmatrix}b\\4\end{bmatrix}\begin{bmatrix}\{p\}\\5\end{bmatrix}\begin{bmatrix}\{c_1\}\\8\end{bmatrix}\begin{bmatrix}b\\9\end{bmatrix}\begin{bmatrix}\{\star_1\}_\end{bmatrix}\begin{bmatrix}\{h_2\}\\6\end{bmatrix}\begin{bmatrix}\{c_2\}\\7\end{bmatrix}\begin{bmatrix}\{\star_2\}_\end{bmatrix},$$

where "_" can be populated with any element of D. Notice the first two symbols of the data word, which do not represent list entries: the first symbol corresponds to the scalar variable k, and the second symbol *nil* is used to model pointer variables that do not reference cells of the lists (in this particular example, there is no such variable). Also notice the two auxiliary "pointer variables" $\star_1, \star_2$, which we use to demarcate the end of the lists. ◁

5.4.2 Converting EQDAs to Strand and the Array Property Fragment

This section presents a translation of an EQDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ into a formula $\varphi_\mathcal{A}$ (in the decidable syntactic fragment of Strand, respectively in the Array Property Fragment) such that the data word language $L_{dat}(\mathcal{A})$ corresponds to the set of program configurations that model $\varphi_\mathcal{A}$.[6] For brevity, we only consider EQDAs working over a single array or list; for multiple lists or arrays, the translation is analogous. We begin this section by introducing some additional definitions and conventions.

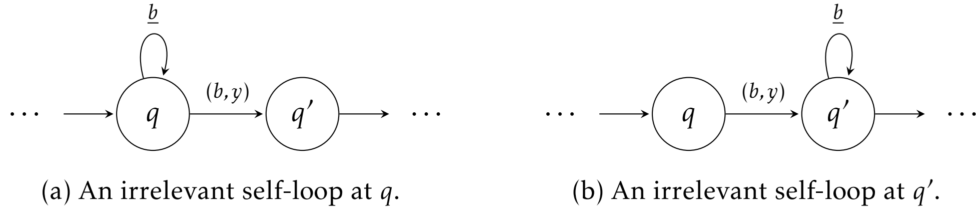
Preliminaries Our translation is based on the notion of *simple paths* in EQDAs. A simple path is a sequence

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

of states connected by transitions starting in the initial state q_0 such that

- $\delta(q_i, a_{i+1}) = q_{i+1}$ is satisfied for all $i \in [n]$;
- no state occurs more than once; and
- all pointer and universally quantified variables occur exactly once in a transition.

[6] In the case of a translation in to the decidable syntactic fragment of Strand, this translation precisely captures the semantics of the considered EQDA, but an approximation might be unavoidable in the case of the Array Property Fragment. We discuss this thoroughly later in this section.



(a) An irrelevant self-loop at q. (b) An irrelevant self-loop at q'.

Figure 5.5: Base cases of the inductive definition of irrelevant self-loops.

Note that all input-symbols a_i that occur in a transition along π are different from $\underline{b}$ because each state is allowed to occur at most once (and we consider EQDAs). For the same reason, there exist only finitely many simple paths in an EQDA. We denote the set of all simple paths of an EQDA $\mathcal{A}$ by $P_{\mathcal{A}}$.

To simplify the translation, we assume without restricting the class of formulas expressible by EQDAs that any EQDA $\mathcal{A}$ fulfills the following three structural properties:

1. Auxiliary variables, such as *nil* or scalar variables, which might have been introduced by the encoding of Section 5.4.1, always occur in the beginning of any simple path in the exact same order. Although the exact order is unimportant, we fix one for the sake of simplicity: scalar variables occur first (in some fixed order), followed by *nil* in the case of EQDAs working over lists, respectively *index_le_zero* and *index_geq_size* in the case of EQDAs working over arrays.

2. Any simple path of $\mathcal{A}$ along which a universally quantified variable occurs together with auxiliary variables leads to a dedicated state labeled with the data formula *true*. This means that the acceptance of a data word depends only on valuations where no universally quantified variable occurs together with auxiliary variables. Since auxiliary variables have been introduced for technical reasons only, valuation words in which a universally variable occurs together with auxiliary variables should, therefore, not influence the formula $\varphi_{\mathcal{A}}$.

3. There are no *irrelevant self-loops* in $\mathcal{A}$, which we define inductively as follows. Let π be a simple path of $\mathcal{A}$, and let q, q' be two states on π such that q' is the direct successor of q and the transition connecting q and q' is $\delta(q, (b, y)) = q'$. If q has a self-loop on $\underline{b}$ (i.e., $\delta(q, \underline{b}) = q$), then we define this self-loop inductively to be *irrelevant on* π if either q' has no self-loop on $\underline{b}$ or if this self-loop is irrelevant on π; the former situation is sketched in Figure 5.5a. Symmetrically, we define a self-loop on $\underline{b}$ at q' inductively as irrelevant on π if either q has no self-loop on $\underline{b}$ or this self-loop is irrelevant on π (see Figure 5.5b).

If a self-loop is irrelevant on π, it cannot contribute to the acceptance of a data word. To see why, consider two valuation words

$$v = \dots (b, y)\underline{b} \dots \text{ and } v' = \dots \underline{b}(b, y) \dots$$

with $\mathrm{dw}(v) = \mathrm{dw}(v')$ (i.e., v and v' only differ in the valuation of the universally quantified variables). Moreover, assume that v is accepted along π using an irrelevant self-loop at q' (see Figure 5.5b). In this situation, $\mathcal{A}$ rejects v' since q has no transition on $\underline{b}$. Thus, $\mathcal{A}$ rejects $\mathrm{dw}(v)$.

The example above shows that one can safely remove irrelevant loops from an EQDA without changing the accepted language of data words. However, since being an irrelevant self-loop is a property depending on a path, it can happen that there are simple paths π, π' such that a self-loop at a state is irrelevant on π but not on π'. To handle such situations, we remove self-loops only temporary: since our translation considers every simple path individually, we remove irrelevant self-loops on the currently processed path only for the sake of translation and restore them once the processing of this path has finished.

In the context of learning EQDAs, one can easily assert the first two properties by constructing a teacher who answers queries accordingly. The actual translation takes care of the third property as described above should it be violated.

EQDAs can check properties of the beginning and the end of a data structure, such as whether a pointer variable points to the head or tail of a list. In order to capture such properties, we use the constants 0 and *size* in the case of arrays, respectively *head* and *tail* in the case of lists, that point to the beginning and the end of the considered data structure. Note that these constants can easily be expressed in both the Array Property Fragment and the syntactic decidable fragment of Strand.

Finally, let us briefly comment on how to express relations between variables (and the constants introduced above). To this end, the Array Property Fragment offers the direct successor relations of array cells, denoted by +1 (used as function; e.g., $x = x' + 1$), the usual less-than relation $<$, and its nonstrict form $\leq$. Similarly, the decidable syntactic fragment of Strand offers the direct successor relation of list cells, denoted by $\rightarrow$, as well as its reflexive closure $\rightarrow^+$ and reflexive and transitive closure $\rightarrow^*$. Note that both +1 and $\rightarrow$ are inelastric relations, whereas the relations $<$, $\leq$, $\rightarrow^+$, and $\rightarrow^*$ are elastic.

Translation Roughly speaking, our translation considers each simple path of an EQDA $\mathcal{A}$ individually, records the structural constraints of the variables along the path, and relates these constraints to the data formula of the final state of the path. By

doing so, we construct a path formula φ_π for each simple path π in $\mathcal{A}$. The resulting formula $\varphi_\mathcal{A}$ is then the union of all such path formulas and an additional subformula that captures the valuation words not accepted by $\mathcal{A}$. Since there exists only finitely many simple path in $\mathcal{A}$, the resulting formula is finite.

More formally, let $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ be a simple path of $\mathcal{A}$ with $a_i \neq \underline{b}$ for $i \in \{1, \ldots, n\}$. The path formula corresponding to π is the implication

$$\varphi_\pi \coloneqq \psi_\pi \rightarrow \chi_\pi,$$

where the antecedent ψ_π (which we define shortly) serves as a guard that captures the relative positions of the variables along π and the consequent $\chi_\pi = f(q_n)$ is the data formula decorating the final state q_n of π (in the case of a translation into the Array Property Fragment, an approximation of both ψ_π and χ_π might be necessary).

We construct the path guard ψ_π as follows: at each state q_i on the simple path, we construct *local constraints*, which describe how individual variables encoded in the incoming and outgoing transitions of q_i are related, and collect them in the set C_i; the path guard then is the conjunction

$$\psi_\pi \coloneqq \bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi.$$

For the construction of path guards, we introduce the following two notations: First, we use $q_{i-1} \xrightarrow{a_i} q_i \in \pi$, respectively $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$, to denote a part of the simple path $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$. Second, we use the input-symbol $a = (\sigma, y) \in \Sigma \times (Y \cup \{-\})$ and the set $(\sigma \cup \{y\}) \setminus \{-\}$ of all variables occurring in a (both pointer variables and universally quantified variables) interchangeably; for instance, we write $x \in a$ to denote that the variable x occurs in a.

We divide the construction of path guards into two parts: The first part (i.e., Cases 1 and 2 below) covers the beginning of the path where pointer variables occur together with auxiliary variables, such as *nil*; recall that our encoding of Section 5.4.1 asserts that auxiliary variables occur always in the beginning of valuation words (and, hence, in simple paths). The second part (i.e., Cases 3 to 6 below) deals with the remainder of the path, which is related to the actual data structure. The local constraints at state q_i are constructed according to the following (nonexclusive) case distinction:

Case 1: $q_{i-1} \xrightarrow{a_i} q_i \in \pi$ *and* $a_i \cap Aux \neq \emptyset$

Let $z \in a_i \cap Aux$ be the unique auxiliary variable.

- If z models a scalar variable, we set $C_i \leftarrow C_i \cup \{x = z\}$ for all $x \in a_i \setminus \{z\}$. (This case covers the second assumed structural property of EQDAs, described on Page 158;

that is, x can only be a universally quantified variable and the state q_n of the simple path is labeled with the data formula *true*.)

- If $z = nil$, we set $C_i \leftarrow C_i \cup \{x = nil\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index_le_size$, we set $C_i \leftarrow C_i \cup \{x < 0\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index_geq_size$, we set $C_i \leftarrow C_i \cup \{x \geq size\}$ for all $x \in a_i \setminus \{z\}$.

Case 2: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$*,* $a_i \cap Aux \neq \emptyset$*, and* $a_{i+1} \cap Aux = \emptyset$

This case covers the boundary between the first and second part of a simple path (i.e., processing the actual data structure starts at q_i). Here, we distinguish two cases:

- If $\delta(q_i, \underline{b})$ is undefined, we set $C_i \leftarrow C_i \cup \{x = head\}$ for all $x \in a_{i+1}$ in the case of lists, respectively $C_i \leftarrow C_i \cup \{x = 0\}$ for all $x \in a_{i+1}$ in the case of arrays.
- If $\delta(q_i, \underline{b}) = q_i$, we set $C_i \leftarrow C_i \cup \{head \rightarrow^* x\}$ for all $x \in a_{i+1}$ in the case of lists, respectively $C_i \leftarrow C_i \cup \{0 \leq x\}$ for all $x \in a_{i+1}$ in the case of arrays.

Cases 3 to 6 below only apply if no auxiliary variables occur in the incoming or outgoing transitions. Note that such situations indeed occur since we assume that Y contains at least one variable (which occurs on every simple path after all auxiliary variables).

Case 3: $q_{i-1} \xrightarrow{a_i} q_i \in \pi$

For all $x, x' \in a_i$ with $x \neq x'$, we set $C_i \leftarrow C_i \cup \{x = x'\}$.

Case 4: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ *and* $\delta(q_i, \underline{b}) = q_i$

Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. In the case of lists, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow^+ x_2\}$. In the case of arrays, we consider two cases:

- If $x_1 \notin Y$ or $x_2 \notin Y$, then we set $C_i \leftarrow C_i \cup \{x_1 < x_2\}$.
- If $x_1 \in Y$, $x_2 \in Y$, and $(a_i \cup a_{i+1}) \cap \Sigma = \emptyset$ (i.e., only universally quantified variables occur), then the Array Property Fragment forbids two adjacent universally quantified variables to be related by the relation $<$; in this case, we set $C_i \leftarrow C_i \cup \{x_1 \leq x_2\}$ and $\chi_\pi \leftarrow \chi_\pi \vee (d(x_1) = d(x_2))$. At this point, the translation does not capture the exact semantics of the EQDA (we comment on this shortly).

Case 5: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ *and* $\delta(q_i, \underline{b})$ *is undefined*

Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. We distinguish two cases:

- If $x_1 \notin Y$ or $x_2 \notin Y$, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow x_2\}$ in the case of lists, respectively $C_i \leftarrow C_i \cup \{x_2 = x_1 + 1\}$ in the case of arrays.

- If $x_1 \in Y$, $x_2 \in Y$, and $(a_i \cup a_{i+1}) \cap \Sigma = \emptyset$, then we need to express the relation of x_1 and x_2 indirectly since both the decidable syntactic fragment of Strand and the Array Property Fragment forbid expressing that two universally quantified variables are a fixed distance away (direct successors in this case). We do so by tracking the distance of x_1, respectively x_2, to either a neighboring pointer variable or one of the constants *head* or *tail*, receptively 0 or *size*. More precisely, we identify a state q on the path π closest to q_i (with respect to the number of transitions) that has a transition containing a pointer variable $p \in PV$, is at the boundary between the first and the second part of π, or is the last state of π. Since $\mathcal{A}$ does not contain any irrelevant self-loops, the subpath from q_i to q has no self-loops. Thus, we can constrain the universally quantified variables at q_i to be a fixed distance away from the pointer variable p or one of the constants (whatever applies). For a translation into the decidable syntactic fragment of Strand, we achieve this by existentially quantifying monadic predicates that track the distance between q and q_i; since this distance is bounded, a finite number of such predicates suffices. For a translation into the Array Property Fragment, we obtain the same effect by means of arithmetic on the pointer variable.

Case 6: $q_{n-1} \xrightarrow{a_n} q_n \in \pi$

In this case, q_n is the last state of π and $\delta(q_{n-1}, a_n) = q_n$ the last transition. We distinguish two cases:

- If $\delta(q_n, \underline{b})$ is undefined, we set $C_n \leftarrow C_n \cup \{x = tail\}$ for all $x \in a_n$ in the case of lists, respectively $C_n \leftarrow C_n \cup \{x = size - 1\}$ for all $x \in a_n$ in the case of arrays.
- If $\delta(q_n, \underline{b}) = q_n$, we set $C_n \leftarrow C_n \cup \{x \rightarrow^* tail\}$ for all $x \in a_n$ in the case of lists, respectively $C_n \leftarrow C_n \cup \{x < size\}$ for all $x \in a_n$ in the case of arrays.

Since the Array Property Fragment lacks the ability to check whether two universally quantified variables are different, Case 4 needs to introduce an overapproximation of the real constraints along a simple path if two universally quantified variables, say y and y', are adjacent at a state with a self-loop on $\underline{b}$ (i.e., the path guard is incorrectly satisfied even if $y = y'$ holds). In order to compensate for this, we amend the formula χ_π by disjointly adding the constraint $d(y) = d(y')$, which ensures that the path formula is satisfied if $y = y'$ holds (since $y = y'$ implies $d(y) = d(y')$). This way, a path formula checks the structural and data constraints of the simple path if the valuation satisfies $y_1 < \cdots < y_k$ (assuming that the predetermined order is $y_1 \prec \cdots \prec y_k$) and is always satisfied if some universally quantified variables are equal. Note that the latter situation cannot be handled by EQDAs because their input alphabet requires universally quantified variables to be at different positions; thus, a path guard should not introduce any constraints in such situations. However, disjointly

adding constraints of the form $d(y) = d(y')$ might be too coarse and makes a path formula with such an approximation imprecise in general.

The complete translation functions as follows: It collects the sets C_i along every simple path $\pi \in P_{\mathcal{A}}$ and constructs the formulas ψ_π and χ_π. For a translation into the decidable syntactic fragment of Strand, it returns the formula

$$\varphi_{\mathcal{A}} \coloneqq \forall y_1 : \ldots \forall y_k : \Bigg[\underbrace{\Bigg(\bigwedge_{\pi \in P_{\mathcal{A}}} \psi_\pi \rightarrow \chi_\pi \Bigg)}_{\varphi_{sp}} \wedge \underbrace{\Bigg(\Bigg[(head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail) \wedge \neg \Bigg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi \Bigg) \Bigg] \rightarrow false \Bigg)}_{\varphi_{\neg sp}} \Bigg];$$

for a translation into the Array Property Fragment, it returns

$$\varphi_{\mathcal{A}} \coloneqq \forall y_1 : \ldots \forall y_k : \Bigg[\underbrace{\Bigg(\bigwedge_{\pi \in P_{\mathcal{A}}} \psi_\pi \rightarrow \chi_\pi \Bigg)}_{\varphi_{sp}} \wedge \underbrace{\Bigg(\Bigg[(0 \leq y_1 \leq \cdots \leq y_k < size) \wedge \neg \Bigg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi \Bigg) \Bigg] \rightarrow \bigvee_{y \neq y' \in Y} d(y) = d(y') \Bigg)}_{\varphi_{\neg sp}} \Bigg].$$

The subformula φ_{sp} is the conjunction of all path formulas, whereas the subformula $\varphi_{\neg sp}$ captures valuation words that have the right ordering of the universally quantified variables but do not admit a run of $\mathcal{A}$ (i.e., that are rejected by $\mathcal{A}$). As in the case of path formulas, the Array Property Fragment formula $\varphi_{\neg sp}$ only approximates the correct semantics of $\mathcal{A}$. Again, the disjunction constituting the consequent compensates for the necessary overapproximation in the antecedent ($y_1 \leq \cdots \leq y_k$ instead of $y_1 < \cdots < y_k$).

Since the decidable syntactic fragment of Strand allows negating atomic formulas, $\varphi_{\mathcal{A}}$ is in this fragment. Though the Array Property Fragment also allows negation over atomic formulas that relate two pointer variables or a pointer variable and a universally quantified variable, negation of an atomic formula of the form $y \leq y'$ is not allowed (see Section 2.2). However, since we assume both a fixed variable ordering on Y along simple paths and that all paths with a different ordering lead to a state labeled with the formula *true*, we can safely remove subformulas of the form $\neg(y \leq y')$ from

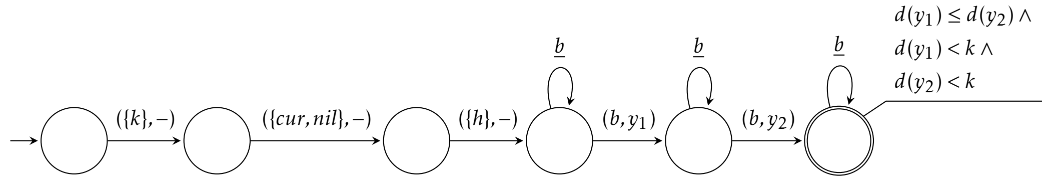


Figure 5.6: A simple path of an EQDA that was learned in our experiments.

$\neg\big(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi\big)$; as before, considering a different ordering of the variables in Y is not necessary because they are universally quantified. After removing such subformulas, the formula $\varphi_\mathcal{A}$ falls into the Array Property Fragment.

The following example illustrates our translation.

Example 5.4. Let us consider the simple path π depicted in Figure 5.6. This simple path is part of an EQDA that was learned in our experiments for the program *list-sorted-find*, which searches for a numerical key k in a sorted list. The program uses the pointer variable h to dereference the head of the list.

The path guard resulting from our translation, which captures the structural constraints along π, is the formula

$$\psi_\pi := \big(cur = nil \wedge h = head \wedge h \rightarrow^+ y_1 \wedge y_1 \rightarrow^+ y_2 \wedge y_2 \rightarrow^* tail\big).$$

Moreover, the path formula is

$$\varphi_\pi := \psi_\pi \rightarrow \big(d(y_1) \le d(y_2) \wedge d(y_1) < k \wedge d(y_2) < k\big).$$ ◁

When applying our translation to an EQDA working over lists, the obtained formula in the decidable syntactic fragment of Strand exactly characterizes the set of program configurations that correspond to the language of data words accepted by the given EQDA. However, due to the abstractions introduced by our translation into the Array Property Fragment, the formula obtained from translating an EQDA over arrays might not characterize the semantics of the given EQDA exactly. Nonetheless, we can at least assert that all data words accepted by this EQDA correspond to a program configuration satisfying the formula.

To make this intuition precise and to avoid cluttering the presentation in the remainder of this section with straightforward details, we loosely introduce the following auxiliary notations: Given a program configuration c, let (c) denote the natural translation of c into an interpretation for formulas in the Array Property Fragment, respectively in the decidable syntactic fragment of Strand (e.g., in the case of the Array Property Fragment, the array is transferred into an uninterpreted function, index

variables become integer variables, and so on).[7] Moreover, let $(c, y_1, \ldots, y_k)$ denote the interpretation (c) in which the universally quantified variables are fixed to the values $y_1, \ldots, y_k$.

The following theorem now summarizes the main result of our translation.

Theorem 5.4. *Let $\mathcal{A}$ be an EQDA, c a program configuration, and u_c the data word corresponding to c. Moreover, let $\varphi_{\mathcal{A}}$ be the formula obtained from the translation described above (either in the decidable syntactic fragment of Strand or in the Array Property Fragment).*

(a) For a translation into the decidable syntactic fragment of Strand, the equivalence

$$u_c \in L_{dat}(\mathcal{A}) \text{ if and only if } (c) \models \varphi_{\mathcal{A}}$$

holds.

(b) For a translation into the Array Property Fragment, the implication

$$u_c \in L_{dat}(\mathcal{A}) \text{ implies } (c) \models \varphi_{\mathcal{A}}$$

holds.

The abstraction along simple paths with $y < y'$ introduced by our translation is the reason why Theorem 5.4 only holds in one direction for the Array Property Fragment (though our experiments, which we discuss in Section 5.6, showed that we nevertheless learned correct invariants for programs manipulating arrays). For this reason, we first prove Part (a) of Theorem 5.4; based on the insight gained in the proof, it becomes much easier to prove Part (b).

Decidable syntactic fragment of Strand The pivotal fact on which Theorem 5.4 relies is that the path guard ψ_π exactly captures the structural constraints along π. The next lemma formalizes this intuition.

Lemma 5.5. *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the decidable syntactic fragment of Strand. Moreover, let c be a program configuration, $y_1, \ldots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \ldots, y_k$. Then, the following equivalence holds:*

the unique run of $\mathcal{A}$ on v is along π if and only if $(c, y_1, \ldots, y_k) \models \psi_\pi$.

[7] We make sure that the type of an interpretation (i.e., whether it is for formulas in the Array Property Fragment or in the decidable syntactic fragment of Strand) is always clear from the context.

Proof of Lemma 5.5. We split the proof into two parts: we first show the direction from left to right and subsequently the reverse direction. The direction from left to right is straightforward and simply exploits the fact that we only add local constraints to a path guard that are obviously satisfied along the given path. The direction from right to left, however, is more elaborate to prove.

From left to right Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

a simple path in $\mathcal{A}$ and assume that the unique run of $\mathcal{A}$ on v is along π. Since the path guard is the conjunction $\bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi$ consisting of all local constraints along π, it is enough to prove that $(c, y_1, \dots, y_k)$ satisfies each individual local constraint. To this end, let ψ be a local constraint, say constructed at state q_i of π.

In order to show that $(c, y_1, \dots, y_k) \models \psi$ holds, we distinguish due to which case of the translation the constraint ψ has been constructed. However, since one can prove most cases using similar arguments, we skip a thorough proof here and exemplary consider Case 4.

If ψ has been introduced in Case 4, then $\psi \coloneqq x_1 \to^+ x_2$ where $x_1 \in a_i$ and $x_2 \in a_{i+1}$. Since the run of $\mathcal{A}$ on v is along the simple path π, we know that all variables $x \in a_i$ occur before the variables $x' \in a_i$. Thus, $(c, y_1, \dots, y_k)$ satisfies $x \to^+ x'$ for all such x, x'. This in turn means $(c, y_1, \dots, y_k) \models \psi$.

From right to left Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} q_m$$

be a simple path in $\mathcal{A}$ and $(c, y_1, \dots, y_k)$ a model of ψ_π. Towards a contradiction, assume that the run of $\mathcal{A}$ on v is along a different simple path, say

$$\pi' = q_0 \xrightarrow{a'_1} q'_1 \xrightarrow{a'_2} \dots \xrightarrow{a'_n} q'_n.$$

Then, there exists a position $i \in \mathbb{N}_+$ at which both paths diverge; that is, $a_j = a'_j$ and $q_j = q'_j$ for all $j \in [i]$, $a_i \neq a'_i$, and $q_i \neq q'_i$. Note that such a position always exists because the states of $\mathcal{A}$ are "typed" (i.e., $\mathcal{A}$ has to remember which variable it has already read). Figure 5.7 depicts such a situation.

We observe that all input-symbols along the paths π and π' are different from $\underline{b}$ because $\mathcal{A}$ is elastic. Thus, if $a_i \neq a'_i$, then there exists a variable $x \in PV \cup Y$ that is missing in exactly one of a_i and a'_i (i.e., $x \in a_i$ if and only if $x \notin a'_i$). Without loss of generality, let us assume $x \in a_i$ and $x \notin a'_i$.

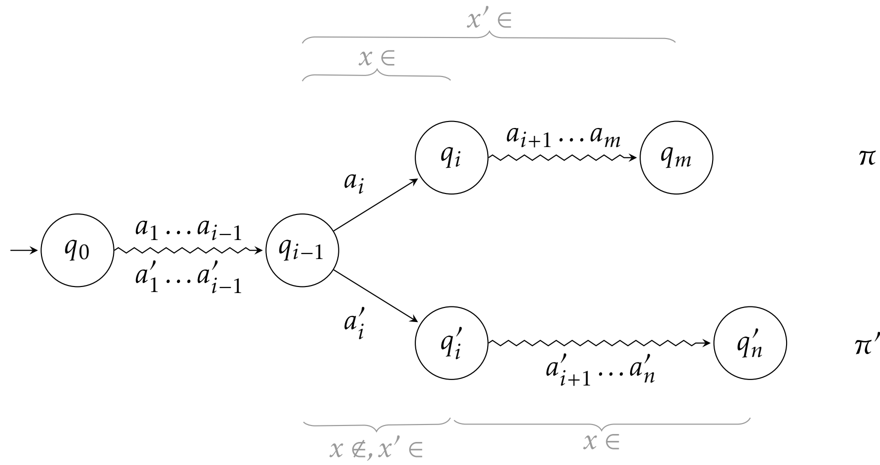


Figure 5.7: Two diverging simple paths.

Since $a'_i \neq \underline{b}$, there also exists a variable $x' \in a'_i$ that is different from x. Moreover, since π' is a simple path (which implies that all pointer and universally quantified variables occur exactly once), the variable x also occurs in π', but only in one of the inputs $a'_{i+1}, \dots, a'_n$; note that x' might or might not occur together with x on π.

We now distinguish two cases:

1. An auxiliary variable, say z, occurs in the input-symbol a_i on π; that is, q_i belongs to the first part of π. We first observe that x cannot be an auxiliary variable because we assume that auxiliary variables appear never together and always in the same, fixed order. Thus, the following two cases remain:
   a) The variable x occurs on the simple path π' together with an auxiliary variable, say z', that is different from z. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models x = z'$. Thus, $(c, y_1, \dots, y_k) \not\models x = z$ because $x = z \wedge x = z'$ is unsatisfiable if $z \neq z'$. However, the path guard ψ_π contains the local constraint $x = z$ (see Case 1). Hence, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.
   b) The variable x does not occur together with an auxiliary variable on the simple path π'. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models \mathit{head} \to^* x$. Consequently, $(c, y_1, \dots, y_k) \not\models x = z$ because z is an auxiliary variable that occurs before *head*. However, the path guard ψ_π contains the local constraint $x = z$ (again, see Case 1). Therefore, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.
2. The input-symbol a_i on π does not contain an auxiliary variable; that is, q_i belongs to the second part of π. Since we assume the run of $\mathcal{A}$ on v to be along

the path π', the variable x' points to a cell that is located before the cell pointed to by x. Hence, $(c, y_1, \dots, y_k) \models x' \to^+ x$. Consequently, $(c, y_1, \dots, y_k) \not\models x \to^* x'$ because $x \to^* x' \wedge x' \to^+ x$ is unsatisfiable. However, the path guard ψ_π implies $x \to^* x'$ (see Cases 4 and 5) although it might not contain this subformula explicitly. Thus, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields the desired contradiction.

This proves that the unique run of $\mathcal{A}$ on v is along the simple path π. □

Using Lemma 5.5, we can now prove Part (a) of Theorem 5.4.

Proof of Theorem 5.4(a). Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \dots \prec y_k$ the predetermined order in which the universally quantified variables have to occur in the input of $\mathcal{A}$, and $\varphi_\mathcal{A}$ the formula in the decidable syntactic fragment of Strand resulting from our translation. In addition, let c be a program configuration and u_c the data word modeling c.

We first show the direction from left to right (i.e., $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$) and subsequently the reverse direction (i.e., $(c) \models \varphi_\mathcal{A}$ implies $u_c \in L_{dat}(\mathcal{A})$).

From left to right Let $u_c \in L_{dat}(\mathcal{A})$. In order to prove that the interpretation (c) satisfies $\varphi_\mathcal{A} \coloneqq \forall y_1 \colon \dots \forall y_k \colon (\varphi_{sp} \wedge \varphi_{\neg sp})$, we fix an arbitrary valuation $y_1, \dots, y_k$ of Y and show

$$(c, y_1, \dots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}.$$

In the case that $head \to^* y_1 \to^+ \cdots \to^+ y_k \to^* tail$ does not hold, we first observe that $(c, y_1, \dots, y_k)$ does not satisfy any path guard because each path guard implies $head \to^* y_1 \to^+ \cdots \to^+ y_k \to^* tail$. Hence, $(c, y_1, \dots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \dots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \dots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In the case that $head \to^* y_1 \to^+ \cdots \to^+ y_k \to^* tail$ holds, let v be the valuation word resulting from extending u_c with the valuation $y_1, \dots, y_k$ (which implies $\mathrm{dw}(v) = u_c$). We proceed the proof by first showing that $(c, y_1, \dots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

1. Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. This particularly means that the unique run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \models f(q)$. By Lemma 5.5, we know $(c, y_1, \dots, y_k) \models \psi_\pi$. Moreover, since $f(q) = \chi_\pi$ and $r \models f(q)$, we also know $(c, y_1, \dots, y_k) \models \chi_\pi$ and, thus, $(c, y_1, \dots, y_k) \models \psi_\pi \to \chi_\pi$. On the other hand, Lemma 5.5 asserts that no other path guard is satisfied by $(c, y_1, \dots, y_k)$. Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp}$.

2. The fact that $(c, y_1, \dots, y_k) \models \psi_\pi$ holds (see the arguments above) immediately shows $(c, y_1, \dots, y_k) \not\models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Hence, the antecedent of $\varphi_{\neg sp}$ is not satisfied and, therefore, $(c, y_1, \dots, y_k) \models \varphi_{\neg sp}$.

Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp} \land \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$.

From right to left To prove this direction, we show that $u_c \notin L_{dat}(\mathcal{A})$ implies $c \not\models \varphi_\mathcal{A}$. To this end, let c be a program configuration and u_c the corresponding data word such that $u_c \notin L_{dat}(\mathcal{A})$.

Since $u_c \notin L_{dat}(\mathcal{A})$, there exists a valuation $y_1, \dots, y_k$ and a corresponding valuation word v (i.e., u_c extended by $y_1, \dots, y_k$ results in v) such that $v \notin L_{val}(\mathcal{A})$. This valuation word is rejected either

1. due to a missing transition; or

2. due to the fact that the run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \not\models f(q)$.

In the first case, the run of $\mathcal{A}$ on v does not lead along a simple path. By Lemma 5.5, this implies $(c, y_1, \dots, y_k) \not\models \psi_\pi$ for every $\pi \in P_\mathcal{A}$. Hence, $(c, y_1, \dots, y_k) \models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Since we assume that $\mathcal{A}$ accepts all valuation words that violate the fixed order of the universally quantified variables or where at least one of these variables points to *nil*, we know that $(c, y_1, \dots, y_k) \models head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail$ holds. Thus, $(c, y_1, \dots, y_k) \not\models \varphi_{\neg sp}$ and, consequently, $c \not\models \varphi_\mathcal{A}$.

In the second case, the run of $\mathcal{A}$ on v leads along a simple path, say π, ending in the configuration (q, r). By Lemma 5.5, this implies $(c, y_1, \dots, y_k) \models \psi_\pi$. However, since $r \not\models f(q) = \chi_\pi$, we have $(c, y_1, \dots, y_k) \not\models \chi_\pi$. Thus, $(c, y_1, \dots, y_k) \not\models \varphi_{sp}$ (because $(c, y_1, \dots, y_k) \not\models \psi_\pi \rightarrow \chi_\pi$) and, consequently, $c \not\models \varphi_\mathcal{A}$.

In total, $u_c \notin L_{dat}(\mathcal{A})$ implies $(c) \not\models \varphi_\mathcal{A}$ (i.e., $(c) \models \varphi_\mathcal{A}$ implies $u_c \in L_{dat}(\mathcal{A})$). ☐

Array Property Fragment The approximation in Case 4 of our translation is the reason why Theorem 5.4 holds only in one direction in the case of a translation into the Array Property Fragment. In order to prove this direction, we first show that the path guard ψ_π overapproximates the structural constraints of π. The next lemma formalizes this.

Lemma 5.6. *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the Array Property Fragment. Moreover, let c be a program*

configuration, $y_1, \dots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \dots, y_k$. Then, the following implication holds:

if the unique run of $\mathcal{A}$ on v is along π, then $(c, y_1, \dots, y_k) \models \psi_\pi$.

Proof of Lemma 5.6. One can prove Lemma 5.6 in the same way as Lemma 5.5 (see Page 165): again, we consider each local constraint ψ of a path guard individually and show $(c, y_1, \dots, y_k) \models \psi$. In fact, we can reuse the proof of Lemma 5.5 except for a slightly different treatment of Case 4, which we sketch below.

Assume that ψ has been added at state q_i of the simple path $\pi = q_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} q_n$, and let $x_1 \in a_i$ and $x_2 \in a_{i+1}$.

If $x_1 \notin Y$ or $x_2 \notin Y$, then this situation matches Case 4 of the proof of Lemma 5.5 and immediately yields the desired result.

If $x_1 \in Y$, $x_2 \in Y$, and both variables do not occur together with a pointer variable, then the translation adds $\psi := x_1 \leq x_2$ instead of the "correct" constraint $x_1 < x_2$. However, we know that all variables $x \in a_i$ appear before the variables $x' \in a_{i+1}$ because the run of $\mathcal{A}$ on v is along π. Thus, $(c, y_1, \dots, y_k) \models x < x'$ for all such x, x', which implies $(c, y_1, \dots, y_k) \models x_1 \leq x_2$ (i.e., $(c, y_1, \dots, y_k) \models \psi$). □

We can now prove Part (b) of Theorem 5.4.

Proof of Theorem 5.4(b). Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \dots \prec y_k$ the predetermined order in which the universally quantified variables have to occur in the input of $\mathcal{A}$, and $\varphi_\mathcal{A}$ the formula in the Array Property Fragment resulting from our translation. Moreover, let c be a program configuration and u_c the data word modeling c. Finally, assume $u_c \in L_{dat}(\mathcal{A})$.

We have to show that (c) is a model of $\varphi_\mathcal{A}$. This proof is similar to the direction from left to right of the proof of Theorem 5.4(a): we again fix an arbitrary valuation $y_1, \dots, y_k$ of Y and show

$$(c, y_1, \dots, y_k) \models \varphi_{sp} \land \varphi_{\neg sp}.$$

In the case that $0 \leq y_1 \leq \dots \leq y_k < size$ does not hold, we observe that $(c, y_1, \dots, y_k)$ does not satisfy any path guard because each path guard implies $0 \leq y_1 \leq \dots \leq y_k < size$. Hence, $(c, y_1, \dots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \dots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \dots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp} \land \varphi_{\neg sp}$.

In the case that $0 \leq y_1 \leq \dots \leq y_k < size$ holds, we distinguish two cases:

1. All universally quantified variables are different; that is, $y_i \neq y_j$ holds for all $i, j \in \{1, \dots, k\}$ with $i \neq j$. In this case, let v be the valuation word resulting from

extending u_c with the valuation $y_1, \dots, y_k$. We proceed the proof by first showing that $(c, y_1, \dots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

a) Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. By Lemma 5.6, we know that then $(c, y_1, \dots, y_k) \models \psi_\pi$ holds. Since $v \in L_{val}(\mathcal{A})$, the registers satisfy the data formula of the final state of π. Thus, $(c, y_1, \dots, y_k) \models \chi_\pi$ and, consequently, $(c, y_1, \dots, y_k) \models \psi_\pi \rightarrow \chi_\pi$.

   To complete this case, we argue that there exists no other path $\pi' \in P_\mathcal{A}$ with $\pi' \neq \pi$ and $(c, y_1, \dots, y_k) \models \psi_{\pi'}$. Towards a contradiction, assume the contrary and let π' such a simple path. By using arguments similar to those in the direction from right to left of the proof of Lemma 5.5, one can show that this can only happen due to an overapproximation of the form $y_i \leq y_j$ (rather than $y_i < y_j$). This, in turn, implies that there exists $i, j \in \{1, \dots, k\}$ with $i < j$ and $y_i = y_k$, which contradicts the assumption that all universally quantified variables are different.

   In total, $(c, y_1, \dots, y_k)$ satisfies the path formula of each simple path. Hence, $(c, y_1, \dots, y_k) \models \varphi_{sp}$.

b) Since $u_c \in L_{dat}(\mathcal{A})$, we know that there exists a simple path $\pi \in P_\mathcal{A}$ such that $(c, y_1, \dots, y_k) \models \psi_\pi$ (see above). Therefore, $(c, y_1, \dots, y_k) \not\models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$ because removing subformulas of the form $\neg(y \leq y')$ from a path guard potentially results in more interpretations satisfying it and, thus, less satisfying its negation. This implies $(c, y_1, \dots, y_k) \models \varphi_{\neg sp}$ since we assume $0 \leq y_1 \leq \dots \leq y_k < size$.

Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

2. There exist $i, j \in \{1, \dots, k\}$ such that $i < j$ and $y_i = y_j$. In this case, there might be a simple path $\pi \in P_\mathcal{A}$ such that $(c, y_1, \dots, y_k) \models \psi_\pi$. Since universally quantified variables never occur together on a simple path (due to the choice of the input alphabet of QDAs), $(c, y_1, \dots, y_k)$ can only satisfy ψ_π due to the overapproximation $y_i \leq y_j$ (rather than $y_i < y_j$) introduced by Case 4 of our translation. This means that the formula χ_π is constructed by taking the disjunction of the formulas $f(q)$ (assuming that q is the final state of π), $d(y_i) = d(y_j)$, and potentially other formulas of the form $d(y) = d(y')$ for $y, y' \in Y$. Thus, $d(y_i) = d(y_j)$ implies χ_π. Since $y_i = y_j$, we have $d(y_i) = d(y_j)$ and, hence, $(c, y_1, \dots, y_k) \models \chi_\pi$. This, in turn, means $(c, y_1, \dots, y_k) \models \psi_\pi \rightarrow \chi_\pi$. Since these arguments are true for all simple paths $\pi' \in P_\mathcal{A}$ for which $(c, y_1, \dots, y_k) \models \psi_{\pi'}$ holds, we have shown $(c, y_1, \dots, y_k) \models \varphi_{sp}$.

On the other hand, $(c, y_1, \dots, y_k) \models \varphi_{\neg sp}$ because $(c, y_1, \dots, y_k)$ satisfies the consequent of $\varphi_{\neg sp}$ due to the equality $y_i = y_j$. Thus, $(c, y_1, \dots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_{\mathcal{A}}$. □

5.5 Learning Universally Quantified Invariants

We embed the active learning algorithm for QDAs, described in Section 5.2, in a passive learning setup in order to learn universally quantified invariants of programs manipulating arrays or lists. Our procedure consists of two successive steps:

1. *Constructing a teacher* We collect a finite sample $\mathcal{S}$ consisting of formula words derived from configurations that manifest on dynamic runs of the program at hand. Then, we construct a teacher who answers queries with respect to $\mathcal{S}$ and ensures that the learned QDA (at least) classifies $\mathcal{S}$ correctly (i.e., the QDA has to be correct at least on the observed behavior of the program).

2. *Running a learner* We plug in our active learning algorithm for QDAs of Section 5.2.2. Once the learning algorithm terminates and returns a QDA, we elastify it if necessary, convert it into a Strand formula or a formula in the Array Property Fragment (whatever is applicable), and return the resulting formula.

The underlying idea of this approach is to pit our learner for QDAs against the teacher sketched above in order to learn an invariant despite the fact that the teacher does not know the invariant himself. In this setting, the learner's objective is to learn a "simplest" QDA that captures the knowledge the teacher possesses, thereby hoping that the learned QDA represents a valid invariant. This approach relies on two key properties: The first is Occam's razor, namely that a simplest set of formula words—represented by a QDA with the least number of states—that is consistent with the observed behavior of the program is a likely invariant. The second is the fact that invariants do often not depend on the actual size and data of data structures. Therefore, a collection consisting of "short" data structures over a "small" finite abstraction of the data domain often contains sufficient information about an invariant.

Our motivation for embedding an active learning algorithm in a passive learning setting is to exploit the strengths of active learning algorithms, such as their polynomial runtime, their ability to generalize information, and so on, whilst taking into account that building a precise teacher is very complex or impossible (as the teacher does not know an invariant and needs to reason about the program at hand). However, active learning algorithms typically ask "short" membership queries (linear in the diameter of the canonical automaton) and produce "small" conjectures. Thus, an imprecise teacher with—explicit or implicit—knowledge about a sufficiently large sample of

(short) formula words is likely to answer most, if not all, queries actually asked by the learner precisely. It is important to emphasize that the learner does not, in general, simply learn a QDA that captures precisely the finite knowledge of the teacher as the representation of this knowledge is often far more complex than a true invariant; instead, he learns a simplest QDA that generalizes the partial knowledge the teacher possesses. This is the reason why our algorithm might not work in time polynomial in the size of a minimal QDA representing an invariant (though it is polynomial in the QDA it finally learns).

One should note at this point that we cannot guarantee that the learned QDA indeed represents a valid invariant as passive learning does not offer a feedback mechanism to correct a wrong conjecture. One possibility to overcome this problem is to repeat the learning procedure iteratively: whenever a conjecture is found to be invalid (e.g., by means of a logic solver), one restarts the learning with a different sample, say extended to include the spurious counterexample or longer data structures over a finer abstraction of the data domain, until a true invariant has been identified. Note, however, that even an iterative algorithm cannot guarantee to converge to an invariant due to the undecidability of the model checking problem.

The following two sections describe how to realize a teacher and a learner for the invariant learning setting described above. Thereby, we assume that the teacher is given a set of formula words that has been extracted from the program in question (we explain how one can do this when describing our experiments). Note that this assumption entails that the formula lattice has been fixed.

Implementing a Teacher

Based on a sample $\mathcal{S}$ of formula words, the teacher answers queries (in accordance with Section 5.2.2) as follows.

Membership query On a membership query with a symbolic word $w \in \Pi^*$, the teacher checks whether $\mathcal{S}$ contains a formula word (w, φ) and returns φ if so. If no such word exists, the teacher returns the formula *false*. Note that returning *false* in the latter case is an arbitrary choice since the teacher effectively does not know how to classify w.

Equivalence query On an equivalence query with a QDA $\mathcal{A}$, the teacher checks whether $\mathcal{A}$ correctly classifies all formula words contained in $\mathcal{S}$ (i.e., he checks whether $(w, \varphi) \in L_{for}(\mathcal{A})$ holds for all $(w, \varphi) \in \mathcal{S}$). If this is the case, the teacher returns "yes". If this is not the case, then there exists a symbolic word $w \in \Pi^*$ such that $(w, \varphi) \in \mathcal{S}$ and $(w, \varphi) \notin L_{for}(\mathcal{A})$ (i.e., there exists a formula $\varphi' \neq \varphi$ with $(w, \varphi') \in L_{for}(\mathcal{A})$), and the teacher returns w as a counterexample.

The construction of such a teacher is straightforward. Note that the teacher might err on queries and, thus, the learned QDA might not be of minimal size. However, the teacher guarantees that the learned QDA at least classifies $\mathcal{S}$ correctly.

Implementing a Learner

The learner does not need much explanation as any active learning algorithm for Moore machines can be employed. On top of that, the learner needs to obtain the input alphabet from the teacher, which depends on the number of pointer variables and universally quantified variables used to build the sample, before the learning can start. Henceforth, the learner queries the teacher until a conjecture passes an equivalence query. If the learned QDA is not elastic, the learner elastifies it as described in Section 5.3 to obtain the unique EQDA that over-approximates the learned QDA. Then, the learner converts this EQDA to a formula in the decidable syntactic fragment of Strand or the Array Property Fragment according to Section 5.4.2 (whatever is applicable), which it finally returns as the result of the learning process.

5.6 Experiments and Evaluation

We implemented a prototype of the teacher and learner presented in Section 5.5 to evaluate the effectiveness and performance of our approach on real-world examples.

Methodology We implemented our prototype in C++ based on the Libalf automata learning framework. The learner uses Rivest and Schapire's learning algorithm for Moore machines, which is natively supported by Libalf. Since Libalf supports arbitrary C++ objects as output of Moore machines, the modifications necessary to adapt Libalf's implementation to the QDA setting were only minor.

In order to extract a sample of formula words from a program, we employed the following three-step procedure:

1. We fixed the formula lattice $\mathcal{F}$ to be the Cartesian lattice over the binary relations $=$, $<$, and $\leq$ as well as the atoms $d(y)$ and $d(x)$ where y is a universally quantified variable and x is a scalar variable. Note that these relations and also the Cartesian abstraction are sufficient to capture invariants of many interesting programs over arrays and lists, such as sorting routines, scanning and searching arrays and lists, in-place reversal of sorted lists, and so on.

2. We added instrumentation code to the header of each loop of the considered program in order to the record configurations realized when running the program. Then, we executed the instrumented code with "small" arrays, respectively lists,

as input and recorded the program configuration at the loop headers. In our experiments, we exhaustively ran the instrumented code on all arrays and lists of a small bounded length $\ell \geq 1$ over the data domain $D = [k]$ for a small integer $k \geq 1$. We chose the values of k and ℓ depending on the considered program such that several hundred program configurations were generated.

3. We constructed a sample $\mathcal{S} \subseteq \Pi^* \times F$ of formula words from the set of program configurations. The set of pointer variables and the set of universally qualified variables define the alphabet Π, whereas the set F of formulas is determined by the formula lattice; at this point, we guessed how many universally quantified variables are sufficient to express an invariant (often two were enough). To convert the program configurations into formula words, we first generated a valuation word according to Section 5.4.1 for each valuation of Y and each program configuration (the construction described in Section 5.4.1 can easily be lifted to valuation words). To obtain the formula word corresponding to a valuation word, we recorded the data values at the positions of the variables in the valuation word, constructed the smallest formula in the lattice that was satisfied by the data, and assigned this formula to the symbolic word that constitutes the valuation word. If more than one valuation word resulted in the same symbolic word but with different data formulas, we associated the symbolic word with the join of all those data formulas.

Verifying whether the learned formula constitutes an invariant for the program in question is an important aspect of our evaluation. To do so, we proceeded as follows:

- In the case of programs working over arrays, we manually derived verification conditions from the program in question, turned them together with the learned logic formula $\varphi_{\mathcal{A}}$ into the SMT-Lib format, and used Microsoft's Z3 SMT solver to verify that the derived formula is in fact an invariant, respectively a function precondition. To improve the accuracy of our implementation, we replaced $\varphi_{\neg sp}$ by the more precise formula $[(0 \leq y_1 < \cdots < y_k < size) \wedge \neg(\bigwedge_{\pi \in P_{\mathcal{A}}} \psi_\pi)] \rightarrow false$. Although this formula does not fall in the Array Property Fragment, Z3 was able to handle it in all of our experiments.

- In the case of programs working over lists, we had to perform the check manually due to the lack of a satisfactory implementation of a decision procedure for the decidable syntactic fragment of Strand. We did so by manually deriving one (or more) simple invariants (respectively function preconditions) of the program at hand and translating them into EQDAs. Then, we checked whether the learned EQDA is equivalent to one of these manually generated by comparing the simple

paths of the former EQDA with those of the latter. Although this approach is tedious, it worked out for all EQDAs of our list examples.

To evaluate our prototype, we used two benchmark suites, comprising the programs listed in Tables 5.1 and 5.2, respectively. For the programs of the first suite, listed in Table 5.1, the task was to learn loop invariants; for the programs of the second suite, listed in Table 5.2, the task was to learn function preconditions. The size of these programs varies between 20 and 100 lines of code, except for a sorting routine taken from the GNU core utilities, which comprises 2500 lines of code. Let us briefly introduce these programs.

The program *array-find* searches for a key in an unsorted array by traversing it from beginning to end. The program *array-copy* copies an array, and the program *array-comp* checks whether two arrays are equal. The programs *insertion-sort-inner*, *insertion-sort-outer*, *selection-sort-inner*, and *selection-sort-outer* implement the insertion sort algorithm, respectively selection sort algorithm, over arrays. Both algorithms have nested loops, and the suffixes *inner* and *outer* refer to the respective loops.

The program *list-find* searches for a key in a sorted list, *list-insert* inserts a key into a sorted list such that the resulting list remains sorted, *list-init* initializes all cells of an input list with a given key, and *list-max* returns the maximum of all the data values stored in a list. The next three programs manipulate multiple lists: *list-merge* takes two sorted lists as input and merges the second list into the first list such that sortedness is retained; *list-partition* takes a list and partitions it into two lists such that the first list has data elements which are less than an input key and the second list has the remaining data elements of the input list; *list-reverse* takes a list sorted in increasing order as input and reverses it in-place such that the output list is sorted in decreasing order. The programs *bubble-sort*, *fold-split*, and *quick-sort* are taken from Bouajjani et al. [BDES12]. The program *list-init-complex* sorts an input array using heap sort and then initializes a list with the content of this sorted array.

The methods *lookup_prev* and *add_cachepage* are part of the module *cachePage*, which belongs to a verified-for-security platform for mobile applications [MPX⁺13] and maintains a cache of the recently used disc pages as a priority queue based on a sorted list. The method *sort* is a merge sort implementation, and *insert_sorted* is a method for insertion into a sorted list; both are methods from Glib, which is a low-level C library that forms the basis of the GTK+ toolkit and the Gnome environment. The methods *devres*[8] and *rm_pkey*[9] are methods adapted from the Linux kernel and an Infiniband device driver [KJD⁺10]. Finally, we consider the sortedness property (with respect to

[8]Corresponds to the method `pcim_iounmap` in the Linux kernel `linux/lib/devres.c`.
[9]Corresponds to the InfiniBand device driver at `drivers/infiniband/hw/ipath/ipath_mad.c`.

Table 5.1: Experimental results of our prototype on learning loop invariants. The column "LOC" refers to lines of code, "EQ" to the number of equivalence queries, "MQ" to the number of membership queries, "Size" to the number of states of the learned QDA, "El." to whether elastification was required, and "Inv." to whether the learned QDA represented an invariant.

Experiment	LOC	Test inputs	t_{teacher} in s	EQ	MQ	Size	El.	t_{learn} in s	Inv.
array-find	25	310	0.05	2	121	8	no	0.00	yes
array-copy	25	7380	1.75	2	146	10	no	0.00	yes
array-compare	25	7380	0.51	2	146	10	no	0.00	yes
insertionsort-outer	30	363	0.19	3	305	11	no	0.00	yes
insertionsort-innner	30	363	0.30	7	2893	23	yes	0.01	yes
selectionsort-outer	40	363	0.18	3	306	11	no	0.01	yes
selectionsort-inner	40	363	0.55	9	6638	40	yes	0.05	yes
list-sorted-find	20	111	0.04	6	1683	15	yes	0.01	yes
list-sorted-insert	30	111	0.04	3	1096	20	no	0.01	yes
list-init	20	310	0.07	5	879	10	yes	0.01	yes
list-max	25	363	0.08	7	1608	14	yes	0.00	yes
list-sorted-merge	60	5004	10.50	7	5775	42	no	0.06	yes
list-partition	70	16 395	11.40	10	11 807	38	yes	0.11	yes
list-sorted-reverse	25	27	0.02	2	439	18	no	0.00	yes
list-bubble-sort	40	363	0.19	3	447	12	no	0.01	yes
list-fold-split	35	1815	0.21	2	287	14	no	0.00	yes
list-quick-sort	100	363	0.03	1	37	5	no	0.00	yes
list-init-complex	80	363	0.05	1	57	6	no	0.01	yes
lookup_prev	25	111	0.04	3	1096	20	no	0.01	yes
add_cachepage	40	716	0.19	2	500	14	no	0.01	yes
Glib sort (merge)	55	363	0.04	1	37	5	no	0.00	yes
Glib insert_sorted	50	111	0.04	2	530	15	no	0.01	yes
devres	25	372	0.06	2	121	8	no	0.00	yes
rm_pkey	30	372	0.06	2	121	8	no	0.00	yes
GNU Coreutils sort	2500	1 File	0.00	17	4996	5	yes	0.07	yes

Table 5.2: Experimental results of our prototype on learning function preconditions. The columns "LOC", "EQ", "MQ", "Size", and "El." are as in Table 5.1; the column "Pre." refers to whether the learned QDA represented a function precondition.

Experiment	LOC	Test inputs	t_{teacher} in s	EQ	MQ	Size	El.	t_{learn} in s	Pre.
list-sorted-find	20	111	0.01	1	37	5	no	0.00	yes
list-init	20	310	0.02	1	26	4	no	0.00	yes
list-sorted-merge	60	329	0.06	3	683	19	no	0.01	yes

the method *compare* that compares two lines) of the method *sortlines*, which lies at the heart of the GNU core utility to sort a file.

We conducted all experiments, whose results we present next, on an Intel Core i5 CPU at 2.4 GHz with 6 GiB of RAM running Ubuntu 10.04 LTS.

Results Tables 5.1 and 5.2 show the results of our experiments. The tables report the lines of code of the program (column "LOC"), the number of test inputs used to generate the sample set (column "Test inputs"), the time needed to construct the teacher from the sample in seconds (column "t_{teacher}"), the number of membership and equivalence queries asked by the learner (columns "MQ", respectively "EQ"), the size of the learned QDA (column "Size"), whether elastification was required (column "El."), the runtime of the learner in seconds (column "t_{learn}"), and whether the synthesized formula was an invariant (column "Inv."), respectively function precondition (column "Pre.").

Our prototype finished on all experiments in less than 1 s. The time required to learn an EQDA was less than 0.1 s, except for the program "list-partition" for which the learning took 0.11 s. The set-up time of the teacher was longer, typically by a factor of 10 to 30, but never exceeded 12 s.

It turned out that the learned QDAs were reasonably small (less than 50 states) although the learned automata might not be the smallest ones representing a given sample (due to the inaccuracies of the teacher). The learner asked a moderate amount of queries: between 2 and 17 equivalence queries, respectively between 26 and 11 807 membership queries. Elastification was often not required (as the learned QDAs were already elastic), and all learned EQDAs represented a valid invariant.

Discussion The main result is that our prototype learned valid invariants, respectively function preconditions, for all experiments within less than one second. The speed of our prototype is mainly due to the moderate number of queries asked during the learning, which shows that applying active learning in a passive learning setting is a worthwhile approach. Though several experiments required manual tweaking until an invariant was learned, the experimental results demonstrate that our learning-based approach is an effective means to synthesize invariants of programs manipulating arrays or lists.

SMT solvers are sometimes capable of verifying inelastic invariants (i.e., formulas over inelastic relations) though inelastic relations lead to undecidable decision procedures in general. However, in our experiments, Z3 was not able to verify such formulas without giving extra trigger, thus, suggesting the necessity of the elastification of QDAs.

We also observed that the learned EQDAs were somewhat larger than expected considering the invariants or function preconditions they represent. It turned out that the reason for this is that EQDAs represent many different ways in which program pointers and universally quantified variables can be shuffled across the linear data structure. A heuristic way to avoid this problem and to reduce the size of an EQDA would be to merge paths that represent a different order of the variables but express logically equivalent constraints. Moreover, data formulas can often be simplified by removing or merging redundant subformulas; for instance, $x_1 < x_2 \vee x_1 = x_2$ can be simplified to $x_1 \leq x_2$. We did not explore these optimizations so far, but the lack of such heuristics in our current implementation is not a drawback as far as verification is concerned.

Finally, let us emphasize that the time taken by our technique to learn an invariant, being black-box, largely depends on the complexity of the property but not on the size of the code. This advantage is most evident from the successful application of our technique to the large sorting program of the GNU core utilities.

5.7 Conclusion

We presented a novel technique to learn loop invariants for programs working over linear data structures such as arrays and lists. Our approach is based on interpreting sets of program configurations as languages of data words and representing those as quantified data automata. To learn quantified data automata, we have developed an algorithm that combines abstract interpretation over data domains with Angluin-style regular language learning for inferring the structural properties of arrays and lists. Furthermore, we proved a unique over-approximation theorem using elastic quantified data automata, which allowed us to translate the properties expressed by an elastic quantified data automaton into the Array Property Fragment and the decidable syntactic fragment of Strand (whatever is applicable). We also implemented a prototype to validate our approach and demonstrated that our technique is able to learn invariants for typical programs, such as finding values in arrays and lists, sorting arrays and lists, inserting values in sorted arrays and lists, and so on.

We recently presented an iterative ICE-learning algorithm for EQDAs [GLMN13a, GLMN14]. This algorithm builds upon a modified version of the RPNI passive learning algorithm and uses an SMT solver to derive examples, counterexamples, and implication-counterexamples. Although we successfully used a prototype to learn quantified invariants for the array-manipulating programs presented in Section 5.6, adapting the prototype to a particular program is currently an elaborate and complicated task; moreover, the lack of an implementation of a decision procedure for

the decidable syntactic fragment of Strand prevented us from automatically learning EQDAs for programs working over lists. Thus, developing a less user-driven approach that automates parts of the manual work currently necessary is an important part of future research.

A natural next step is to integrate further optimization into our prototype and to turn it into a robust tool (preferably together with the ICE-learning-based approach mentioned above). Another challenge clearly lies in applying our prototype to large real-world examples.

We demonstrated that learning structural conditions of data structure invariants using automata is an effective technique, especially for quantified properties where direct or machine learning techniques are currently unknown. However, for the data formulas themselves, machine learning can be very effective [SNA12], and exploring a combination of automata-based learning of structural properties and machine learning of data formulas seems to be a promising direction of future research. Note that in our current implementation, the data formulas have to be provided by the user.

Another promising future direction of our work is to learn invariants over tree data structures and other recursive data types for which currently no effective mechanisms are known. The reason for hope in using our framework is that many of the components needed are already in place for trees—tree automata are robust, and effective Angluin-style algorithms for learning them are known. Moreover, a version of Strand over trees that restricts relations over universally quantified variables to be elastic is known to be decidable [MPQ11, MQ11]. However, defining elastic tree automata and implementing a decision procedure for the decidable syntactic fragment of Strand over trees remain great challenges.

6

Automatic Reachability Games

The second part of this thesis addresses the question of how to lift the merits of automata learning techniques to infinite games.

In the present chapter, we focus on *reachability games* in the sense of Section 2.3. Since not much research has been devoted to applying automata learning to infinite games so far, reachability games—as one of the most fundamental types of games—are a natural choice to begin with. Furthermore, many solutions for more complex games computationally rely on solving reachability games; for instance, a (learning-based) algorithm for solving reachability games can be used to solve Büchi games.[1]

In order to apply automata learning techniques to (reachability) games, we first need a proper representation of games. A natural approach, which we pursue in this chapter, is to consider games that are played on automatic graphs. This leads to the definition of *automatic reachability games*: such games consists of two components, an arena whose underlying graph is automatic and a regular set of *target vertices*. Note that this definition does not only allow us to apply automata learning techniques but also makes it possible to deal with both finite and infinite arenas. We introduce automatic reachability games in detail in Section 6.1.

The first major part of this chapter (i.e., Section 6.2) deals with automatic reachability games over *finite arenas*. For this kind of arenas, we present an algorithm based on iteratively computing the attractor by means of DFAs—but not yet using learning. A framework introduced Alur, Madhusudan, and Nam [AMN05b] makes it possible to compare this approach to other symbolic methods such as fixed-point computations based on binary decision diagrams, transformations into Propositional Boolean Logic, and transformations into quantified Propositional Boolean Logic.

[1] We refer the reader to Thomas [Tho95] for a thorough description of how to solve Büchi games by means of solving a series of reachability games.

In the second part (i.e., in Section 6.3), we consider automatic reachability games over *infinite arenas*. In this setting, fixed-point computations (e.g., as usually used in the case of finite arenas) do no longer guarantee to converge within a finite number of iterations. In the context of verification, in particular in Regular Model Checking, various acceleration techniques have been developed and applied successfully to overcome this problem. Perhaps the most prominent example is *widening* (e.g., see Touili [Tou01]). We already discussed some of these techniques in the Related Work section of Chapter 4.

We here follow a different direction and learn the attractor, provided it can be represented as a regular set, instead of computing it iteratively. One possible approach for this would be to try adapting our learning-based techniques for Regular Model Checking of Chapter 4. One might think of two orthogonal ways to do so: The first would be to express the semantics of reachability games (i.e., "alternating reachability" as opposed to "plain reachability" as in the case of Regular Model Checking) in terms of logic formulas and adapt the techniques of Chapter 4 accordingly. The second approach would be to incorporate alternating reachability into the transducer of the given Regular Model Checking instance and apply the techniques of Chapter 4 straight away. As it turns out, however, both approaches seem to be dead ends due to the lack of a satisfactory way to capture the game semantics.

Therefore, we develop a different technique, adapted from Vardhan et al. [VSVA05], which works in Angluin's original setting and allows applying off-the-shelf active learning algorithms. However, since the attractor is a priori unknown—the task is learning it after all—, our approach has the problem that a teacher is unable to answer membership queries (as this would entail to solve the game beforehand). We approach this obstacle by learning not the attractor but the fixed point of a particular functional about which the teacher has complete knowledge and from which one can derive the winning regions as well as positional winning strategies for both players. A teacher for this fixed point can not only answer membership and equivalence queries precisely but also removes the need for coping with "don't know"-answers.

A drawback of this indirect approach is that the fixed point we aim at might not be a regular set even if the attractor is. If this happens to be the case, a learning algorithm (for regular languages) does not terminate. Unfortunately, the question of whether our alternative attractor characterization is regular is undecidable, which we prove later in this chapter. However, it is guaranteed to be regular if the underlying arena is finite. This fact allows applying our learning-based technique also to arbitrary reachability games over finite arenas.

Based on a prototype implementation, we demonstrate the feasibility of both of our techniques in Section 6.3. To assess the performance of our DFA-based attractor computation, we compare our prototype to a reference benchmark by Alur, Madhusu-

dan, and Nam [AMN05b]; this benchmark comprises two reachability games over finite arenas, which can be varied in size. To evaluate our learning-based method, we extended one of these games to an infinite arena.

The results of this chapter partly appeared in conference proceedings [Nei10].

Related Work

To the best of our knowledge, there exists no literature on automata learning in the context of (reachability) games. However, reachability games over both finite and infinite arenas are well-studied.

Reachability games over finite arenas have been comprehensively investigated (e.g., by Thomas and others [Tho95, GTW02]). To solve reachability games, an efficient fixed-point algorithm exists, which computes winning regions and winning strategies in time and space linear in the number of edges of the arena.

Reachability games over infinite arenas have been studied in various contexts. To name but a few examples, Cachat [Cac02] investigated reachability games over pushdown graphs, Brozek [Bro11] considered (stochastic) reachability games over one-counter graphs, and Brázdil, Jančar, and Kučera [BJK10] studied reachability games over extended vector addition systems with states. However, we are not aware of any other work dealing with reachability games over automatic arenas.

Finally, it is worth mentioning that reachability games have also been generalized and extended (e.g., see Fijalkow and Horn [FH10]). Further examples are timed and stochastic games (e.g., by Bouyer and Forejt [BF09]) as well as concurrent games (e.g., by de Alfaro, Henzinger, and Kupferman [dAHK07]).

6.1 Automatic Arenas and Automatic Reachability Games

An automatic reachability game is a reachability game that is played on an arena whose underlying graph is automatic. Moreover, we require for technical reasons that every vertex has a finite number of outgoing edges. We call these particular types of arenas *automatic arenas* and use the following definition.

Definition 6.1 (Automatic arena)**.** An *automatic arena* is an arena $\mathfrak{A} = (V_0, V_1, E)$ in which the graph (V, E) with $V = V_0 \uplus V_1$ is automatic and $E \cap (\{v\} \times V)$ is finite for every $v \in V$.

Note that the vertex set of every automatic graph is countable (since Σ^* is countable for every alphabet Σ) and, hence, Definition 6.1 complies with our definition of arenas, which requires V_0 and V_1 to be countable sets.

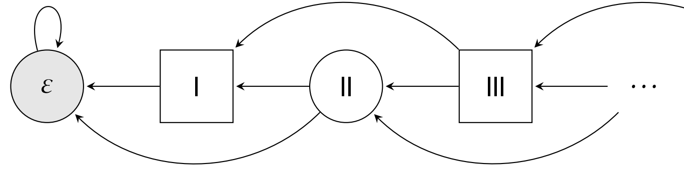


Figure 6.1: The reachability game of Example 6.1. The set F is a singleton set and contains the gray-shaded vertex.

To ease the descriptions in this chapter, let from now on $V = V_0 \cup V_1$. In addition, we assume without loss of generality that $V_0, V_1 \subseteq \Sigma^*$ are regular sets over a fixed alphabet Σ and $E \subseteq \Sigma^* \times \Sigma^*$ is an automatic relation. This means in particular that a vertex $v \in V$ is a (finite) word over the alphabet Σ.

To work with an automatic arena algorithmically, we assume that it is given as a tuple $\mathfrak{A} = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T})$ consisting of

- a DFA $\mathcal{A}_{V_0}$ over Σ accepting V_0;
- a DFA $\mathcal{A}_{V_1}$ over Σ accepting V_1; and
- a synchronous transducer $\mathcal{T}$ over the input and output alphabet Σ defining E.

The definition of automatic reachability games is now straightforward.

Definition 6.2 (Automatic reachability game)**.** An *automatic reachability game* is a reachability game $\mathfrak{G} = (\mathfrak{A}, F)$ where $\mathfrak{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{T})$ is an automatic arena and the set $F \subseteq L(\mathcal{A}_{V_0}) \cup L(\mathcal{A}_{V_1})$ of *target vertices* is regular.

Also here, we assume that F is given as a DFA $\mathcal{A}_F$ and denote an automatic reachability game by $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$.

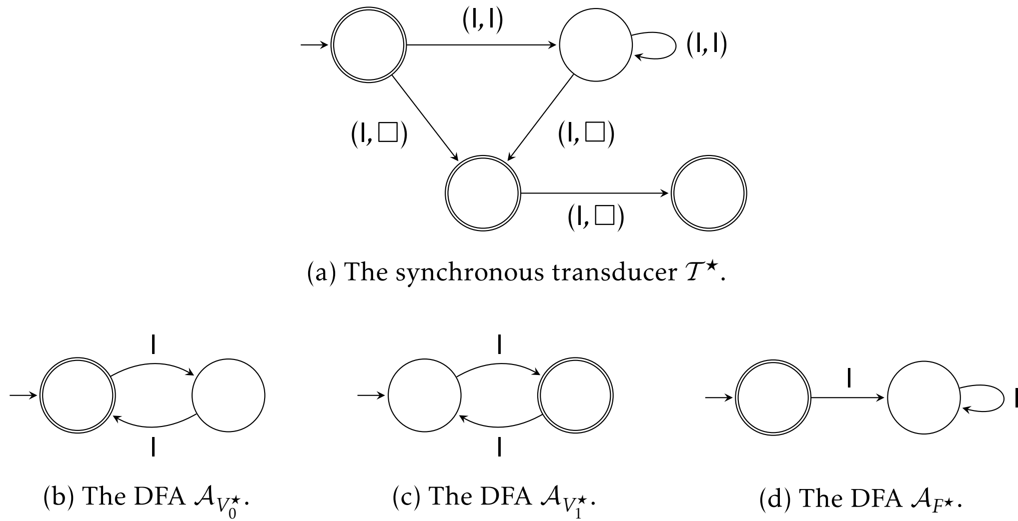
The following example illustrates these definitions.

Example 6.1. Let us consider the reachability game $\mathfrak{G}^\star = (\mathfrak{A}^\star, F^\star)$ over the arena $\mathfrak{A}^\star = (V_0^\star, V_1^\star, E^\star)$ with

- $V_0^\star = \{\mathsf{I}^i \mid i \text{ is even}\}$ and $V_1^\star = \{\mathsf{I}^i \mid i \text{ is odd}\}$;
- $E^\star = \{(\varepsilon, \varepsilon)\} \cup \{(\mathsf{I}^{i+1}, \mathsf{I}^i) \mid i \in \mathbb{N}\} \cup \{(\mathsf{I}^{i+2}, \mathsf{I}^i) \mid i \in \mathbb{N}\}$; and
- $F^\star = \{\varepsilon\}$.

This game is depicted in Figure 6.1.

It is not hard to verify that $\mathfrak{G}^\star$ is an automatic reachability game: Figure 6.2 depicts the DFAs $\mathcal{A}_{V_0^\star}$, $\mathcal{A}_{V_1^\star}$, and $\mathcal{A}_{F^\star}$ as well as the transducer $\mathcal{T}^\star$ that define $\mathfrak{G}^\star$. ◁



(a) The synchronous transducer $\mathcal{T}^\star$.

(b) The DFA $\mathcal{A}_{V_0^\star}$.

(c) The DFA $\mathcal{A}_{V_1^\star}$.

(d) The DFA $\mathcal{A}_{F^\star}$.

Figure 6.2: An automatic representation of the reachability game of Example 6.1.

6.2 Automatic Reachability Games over Finite Arenas

Let us first consider automatic reachability games that are played on a finite arena. For the remainder of this section, we fix an automatic reachability game $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$ with arena $\mathfrak{A} = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T})$. We additionally assume that the vertex set $V = V_0 \cup V_1$ contains a *finite* number of vertices.

To solve $\mathfrak{G}$, we propose a symbolic version of the standard attractor computation described in Section 2.3. As in the case of the standard attractor computation, we start with $\mathrm{Attr}_\sigma^0(F) = F$ and successively compute the sets

$$\mathrm{Attr}_\sigma^{i+1}(F) = \mathrm{Attr}_\sigma^i(F) \cup \mathrm{Pre}_\sigma(\mathrm{Attr}_\sigma^i(F)) \cup \mathrm{Pre}_{1-\sigma}(\mathrm{Attr}_\sigma^i(F))$$

for increasing values of i where

$$\mathrm{Pre}_\sigma(Y) = \{v \in V_\sigma \mid \exists v' \in Y \colon (v, v') \in E\} \text{ and}$$
$$\mathrm{Pre}_{1-\sigma}(Y) = \{v \in V_{1-\sigma} \mid \forall v' \in V \colon (v, v') \in E \rightarrow v' \in Y\}.$$

In order to do so, we carry out each step of this fixed-point computation using DFAs. More precisely, we start with the DFA $\mathcal{A}_0 = \mathcal{A}_F$ and, given a DFA $\mathcal{A}_i$ accepting $\mathrm{Attr}_\sigma^i(F)$, we use $\mathcal{A}_i$ to construct a DFA $\mathcal{A}_{i+1}$ that accepts $\mathrm{Attr}_\sigma^{i+1}(F)$. The next lemma states that this procedure is feasible.

Lemma 6.1. *Let $\mathfrak{G} = (\mathfrak{A}, F)$ be an automatic reachability game. If $\mathrm{Attr}_\sigma^i(F)$ is a regular set, so is $\mathrm{Attr}_\sigma^{i+1}(F)$. Moreover, given a DFA accepting $\mathrm{Attr}_\sigma^i(F)$, one can effectively construct a DFA accepting $\mathrm{Attr}_\sigma^{i+1}(F)$.*

Proof of Lemma 6.1. Since regular languages are closed under finite union, it is enough to provide a construction that takes a DFA accepting some regular set $Y \subseteq V$ and produces two new DFAs that accept $\mathrm{Pre}_\sigma(Y)$ and $\mathrm{Pre}_{1-\sigma}(Y)$, respectively. We then obtain the intended result by setting $Y = \mathrm{Attr}^i_\sigma(F)$.

We split our construction into two parts: we construct a DFA $\mathcal{A}^{\mathrm{pre}}_\sigma$ accepting $\mathrm{Pre}_\sigma(Y)$ in the first part and a DFA $\mathcal{A}^{\mathrm{pre}}_{1-\sigma}$ accepting $\mathrm{Pre}_{1-\sigma}(Y)$ in the second. For the remainder of this proof, let $\mathcal{A}_Y$ be a DFA with $L(\mathcal{A}_Y) = Y$ and $Y \subseteq V$.

We construct the DFA $\mathcal{A}^{\mathrm{pre}}_\sigma$ as follows:

1. We construct an intermediate NFA $\mathcal{A}'$ such that $L(\mathcal{A}') = R(\mathcal{T})^{-1}(L(\mathcal{A}_Y))$ according to Corollary 2.1 (see Page 18); that is, the NFA $\mathcal{A}'$ accepts all predecessor-vertices of vertices in Y.

2. We determinize $\mathcal{A}'$ and construct the DFA $\mathcal{A}^{\mathrm{pre}}_\sigma$ such that $L(\mathcal{A}^{\mathrm{pre}}_\sigma) = L(\mathcal{A}') \cap \mathcal{A}_{V_0}$.

The automaton $\mathcal{A}^{\mathrm{pre}}_\sigma$ resulting from the construction above accepts $\mathrm{Pre}_\sigma(X)$, as intended. To see this, consider the following equivalences:

$$\begin{aligned} v \in L(\mathcal{A}^{\mathrm{pre}}_\sigma) &\Leftrightarrow v \in L(\mathcal{A}_{V_0}) \cap R(\mathcal{T})^{-1}(L(\mathcal{A}_Y)) \\ &\Leftrightarrow v \in V_0 \wedge v \in E^{-1}(Y) \\ &\Leftrightarrow v \in V_0 \wedge \exists v' \in Y \colon (v, v') \in E \\ &\Leftrightarrow v \in \mathrm{Pre}_\sigma(Y). \end{aligned}$$

The construction of $\mathcal{A}^{\mathrm{pre}}_{1-\sigma}$ is similar to the one of $\mathcal{A}^{\mathrm{pre}}_\sigma$ and proceeds as follows:

1. We construct a DFA $\mathcal{A}''$ such that $L(\mathcal{A}'') = V \setminus L(\mathcal{A}_Y)$; that is, $\mathcal{A}''$ accepts all vertices not belonging to Y.

2. We construct an NFA $\mathcal{A}'''$ such that $L(\mathcal{A}''') = R(\mathcal{T})^{-1}(L(\mathcal{A}''))$. This NFA accepts all vertices with at least one successor not belonging to Y.

3. We determinize $\mathcal{A}'''$ and construct $\mathcal{A}^{\mathrm{pre}}_{1-\sigma}$ such that $L(\mathcal{A}^{\mathrm{pre}}_{1-\sigma}) = L(\mathcal{A}_{V_1}) \setminus L(\mathcal{A}''')$.

Finally, we claim that the DFA $\mathcal{A}^{\mathrm{pre}}_{1-\sigma}$ indeed accepts $\mathrm{Pre}_{1-\sigma}(Y)$. The following equivalences prove this:

$$\begin{aligned} v \in L(\mathcal{A}^{\mathrm{pre}}_{1-\sigma}) &\Leftrightarrow v \in V_1 \wedge v \notin R(\mathcal{T})^{-1}(V \setminus L(\mathcal{A}_Y)) \\ &\Leftrightarrow v \in V_1 \wedge \neg\big(\exists v' \in V \colon (v, v') \in E \wedge v' \in V \setminus Y\big) \\ &\Leftrightarrow v \in V_1 \wedge \forall v' \in V \colon (v, v') \in E \rightarrow v' \in Y \\ &\Leftrightarrow v \in \mathrm{Pre}_{1-\sigma}(Y). \end{aligned}$$

□

Algorithm 6.1: DFA-based symbolic attractor computation.

Input: An automatic reachability game $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$ over a finite arena.

1 $\mathcal{A}_0 \leftarrow \mathcal{A}_F$.
2 $i \leftarrow 0$.
3 **repeat**
4 | $\mathcal{A}_{i+1} \leftarrow \mathit{dfaAttr}(\mathcal{A}_i)$.
5 | $i \leftarrow i + 1$.
6 **until** $\mathcal{A}_i = \mathcal{A}_{i-1}$
7 **return** $\mathcal{A}_0, \dots, \mathcal{A}_{i-1}$.

To solve automatic reachability games over finite arenas, we can now apply the symbolic attractor computation shown in Algorithm 6.1: it starts with $\mathcal{A}_0 = \mathcal{A}_F$ (i.e., $L(\mathcal{A}_0) = \mathrm{Attr}_0^0(F)$) and successively computes DFAs $\mathcal{A}_i$ satisfying $L(\mathcal{A}_i) = \mathrm{Attr}_0^i(F)$; the construction of the DFA $\mathcal{A}_{i+1}$ from $\mathcal{A}_i$ is carried out according to Lemma 6.1 by the subprocedure *dfaAttr*. Since the vertex set is assumed to be finite, say containing n vertices, this computation converges after at most $n+1$ steps and results in the sequence $\mathcal{A}_0, \dots, \mathcal{A}_n$ of DFAs with

$$L(\mathcal{A}_n) = \mathrm{Attr}_0(F).$$

Consequently, the winning regions are $W_0 = L(\mathcal{A}_n)$ and $W_1 = (L(\mathcal{A}_{V_0}) \cup L(\mathcal{A}_{V_1})) \setminus W_0$. Moreover, based on the DFAs $\mathcal{A}_0, \dots, \mathcal{A}_n$, we can extract positional winning strategies for both players according to Section 2.3. Note that applying the definitions of Section 2.3 is possible since $\mathrm{Attr}_0^0(F), \dots, \mathrm{Attr}_0^n(F)$ are represented by DFAs, which allows deciding the membership of a vertex in one of these sets. In summary, we obtain the following result.

Theorem 6.2. *Given an automatic reachability game $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$ over a finite arena, say containing n vertices, Algorithm 6.1 terminates after at most $n+1$ iterations and returns DFAs $\mathcal{A}_0, \dots, \mathcal{A}_n$ that satisfy $L(\mathcal{A}_i) = \mathrm{Attr}_0^i(F)$ for $i \in [n+1]$. Based on these DFAs, one can derive the winning regions and positional winning strategies for both players according to Section 2.3.*

We conclude this section with a remark about the time and space complexity of Algorithm 6.1. Since each step of the computation involves determinizing two NFAs, the size of the DFA $\mathcal{A}_{i+1}$ can be exponentially larger than the one of $\mathcal{A}_i$. In the worst case, the size of the final DFA $\mathcal{A}_n$ is a tower of two of height n; that is, our automata-based attractor computation has a nonelementary time and space complexity. However, despite this large theoretical upper bound, we demonstrate in Section 6.4 that our approach performs well in practice and is competitive with other symbolic techniques.

6.3 Automatic Reachability Games over Infinite Arenas

Let us now consider automatic reachability games that are played on a (potentially) infinite arena. Again, we fix an automatic reachability game $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$ with arena $\mathfrak{A} = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T})$.

For arenas with an infinite number of vertices, fixed-point computations (e.g., the one we used in Section 6.2) are no longer guaranteed to converge and terminate in finite time. To overcome this problem, we introduce a learning-based algorithm that aims at actively learning the attractor in interaction with a teacher rather than computing it iteratively. A further beneficial effect is that the performance of a learning-based algorithm does not depend on the size of intermediate results (e.g., as the algorithm presented in the previous section) but on the size of the final result.

The learning takes place in Angluin's original active learning setting (in which the teacher answers membership and equivalence queries). However, actively learning the attractor in the context of automatic reachability games is problematic. In fact, already answering the membership query "Does $v \in \mathrm{Attr}_\sigma(F)$ hold?" is impossible because the reachability problem in automatic graphs, which is known to be undecidable (see Observation 2.2 on Page 19), can be reduced to this question. Therefore, we introduce an alternative characterization of the attractor, using a functional Γ_σ, that subsumes the attractor and allows constructing an appropriate teacher. Once we have constructed such a teacher, it is sufficient to plug in our preferred active learning algorithm and extract the attractor after the learning has finished.

We present our alternative attractor characterization in Section 6.3.1. Subsequently, in Section 6.3.2, we describe how to build a corresponding teacher. Finally, we demonstrate in Section 6.3.3 how to extract winning regions and winning strategies from the learned automaton. We also prove that it is undecidable whether our alternative attractor characterization is regular, and, thus, can be learned by our approach.

6.3.1 An Alternative Characterization of the Attractor

Since learning the attractor is problematic due to the inability of answering membership queries, we move to a setting in which answering queries is possible. Our pivotal idea, which we adapt from Vardhan et al. [VSVA05], is to add supplementary information to the attractor: instead of $\mathrm{Attr}_\sigma(F)$, we learn the augmented set

$$X_\sigma = \{(v, i) \in V \times \mathbb{N} \mid v \in \mathrm{Attr}^i_\sigma(F)\}.$$

The meaning of a pair $(v, i) \in X_\sigma$ is that Player σ can force a play starting in v to visit a vertex in F *in at most i moves*; we encourage the reader to think of i as a distance information that witnesses the membership of v in $\mathrm{Attr}^i_\sigma(F)$.

Clearly, a teacher does not know X_σ in advance and needs other means to answer queries. Our solution is to use a functional $\Gamma_\sigma \colon 2^{V \times \mathbb{N}} \to 2^{V \times \mathbb{N}}$ whose unique fixed point is X_σ and about which a teacher can answer queries. Intuitively, Γ_σ corresponds to one step in the attractor computation of Player σ that additionally takes the distance information into account. Formally, we define Γ_σ by

$$\Gamma_\sigma(Y) = F \times \mathbb{N} \cup \gamma_\sigma(Y) \cup \gamma_{1-\sigma}(Y),$$

where

$$\begin{aligned} \gamma_\sigma(Y) &= \{(v, i+1) \mid v \in V_\sigma \text{ and } \exists v' \in V \colon (v, v') \in E \text{ and } (v', i) \in Y\} \text{ and} \\ \gamma_{1-\sigma}(Y) &= \{(v, i+1) \mid v \in V_{1-\sigma} \text{ and } \forall v' \in V \colon (v, v') \in E \to (v', i) \in Y\}. \end{aligned}$$

It is not hard to verify that Γ_σ is *monotonic* (i.e., $Y \subseteq Y'$ implies $\Gamma_\sigma(Y) \subseteq \Gamma_\sigma(Y')$) and, hence, a fixed point of Γ_σ exists. Moreover, as the next lemma states, every fixed point of Γ_σ is merely an alternative, though complete description of $\mathrm{Attr}_\sigma(F)$. In fact, the lemma implies that Γ_σ has a unique fixed point and that this fixed point coincides with the set X_σ.

Lemma 6.3. *If Y is a fixed point of Γ_σ, then $(v,i) \in Y$ holds if and only if $v \in \mathrm{Attr}^i_\sigma(F)$.*

Proof of Lemma 6.3. Let $Y \subseteq V \times \mathbb{N}$ be a fixed point of Γ_σ and $v \in V$ a vertex. The proof proceeds by induction over the distance information $i \in \mathbb{N}$.

Base case Let $i = 0$. We know by definition of Γ_σ that $(v, 0) \in Y$ holds if and only if $v \in F$. Therefore, the claim immediately holds because $\mathrm{Attr}^0_\sigma(F) = F$.

Induction step Let $i > 0$. We distinguish whether $v \in V_\sigma$ or $v \in V_{1-\sigma}$.

- Let $v \in V_\sigma$. We first show the direction from left to right. To this end, let $(v,i) \in Y$. We now consider two cases: $(v, i-1) \in Y$ and $(v, i-1) \notin Y$. If $(v, i-1) \in Y$, then applying the induction hypothesis yields $v \in \mathrm{Attr}^{i-1}_\sigma(F)$. Thus, also $v \in \mathrm{Attr}^i_\sigma(F)$ holds since $\mathrm{Attr}^{i-1}(F) \subseteq \mathrm{Attr}^i(F)$. If $(v, i-1) \notin Y$, on the other hand, then there exists a $v' \in V$ with $(v, v') \in E$ and $(v', i-1) \in Y$ because Y is a fixed point of Γ_σ. By applying the induction hypothesis, this is true if and only if $v' \in \mathrm{Attr}^{i-1}_\sigma(F)$. Thus, $v \in \mathrm{Attr}^i_\sigma(F)$.

  For the direction from right to left, let $v \in \mathrm{Attr}^i_\sigma(F)$. We again consider two cases: $v \in F$ and $v \notin F$. If $v \in F$, then $(v, i) \in Y$ immediately holds since $F \times \mathbb{N} \subseteq \Gamma_\sigma(Y)$ for any $Y \subseteq V \times \mathbb{N}$. If $v \notin F$, then there exists a $v \in V$ with $(v, v') \in E$ and $v' \in \mathrm{Attr}^{i-1}_\sigma(F)$. Applying the induction hypothesis then yields $(v', i-1) \in Y$. Thus, $(v, i) \in Y$ because Y is a fixed point of Γ_σ.

- The case $v \in V_{1-\sigma}$ is similar; hence, let us just prove the claim for the direction from left to right. To this end, let $(v,i) \in Y$. If $(v,i-1) \in Y$, then applying the induction hypothesis yields $v \in \mathrm{Attr}_\sigma^{i-1}(F)$ and, therefore, also $v \in \mathrm{Attr}_\sigma^{i}(F)$. If $(v,i-1) \notin Y$, then $(v',i-1) \in Y$ for all $v' \in V$ with $(v,v') \in E$ since Y is a fixed point of Γ_σ. By induction hypothesis, this is true if and only if $v' \in \mathrm{Attr}_\sigma^{i-1}(F)$ holds for all such v'. Hence, $v \in \mathrm{Attr}_\sigma^{i}(F)$. □

Corollary 6.1. The unique fixed point of Γ_σ is X_σ.

In order to actively learn the set X_σ, we need to fix an encoding of pairs that is amenable to automata learning techniques. Formally, we define an *encoding of pairs* to be a computable bijective mapping $enc\colon V \times \mathbb{N} \to \Sigma_{enc^*}$, which uniquely encodes the pair $(v,i) \in V \times \mathbb{N}$ as the finite word $enc(v,i) \in \Sigma_{enc^*}$. We lift encodings to sets of pairs $Y \subseteq V \times \mathbb{N}$ in the usual way (i.e., $enc(Y) = \{enc(v,i) \mid (v,i) \in Y\}$).

We now introduce an encoding, which we call enc^*, that allows us to construct a teacher for the set X_σ. Intuitively, enc^* encodes a pair $(v,i) \in V \times \mathbb{N}$ as the convolution[2] of the word v and a unary encoding of i. Formally, we encode the natural number $i \in \mathbb{N}$ as the word I^i over the alphabet $\Sigma_D = \{\mathsf{I}\}$. Letting $\Sigma_{enc^*} = (\Sigma \cup \{\Box\}) \times (\Sigma_D \cup \{\Box\})$ be the encoding alphabet—where Σ is the alphabet of the game $\mathfrak{G}$ and $\Box \notin \Sigma \cup \Sigma_D$ is a new padding symbol—we define

$$enc^*(v,i) = v \otimes \mathsf{I}^i \in \Sigma_{enc^*}^*.$$

Note, however, that this particular encoding is but one choice amongst several that allow us to construct a teacher for automatic reachability games.

The next example illustrates the encoding enc^*.

Example 6.2. Given the alphabet $\Sigma = \{a,b\}$, the encoding of the pair $(aba,5)$ with respect to enc^* is the word

$$enc^*(aba,5) = \begin{bmatrix} a \\ \mathsf{I} \end{bmatrix}\begin{bmatrix} b \\ \mathsf{I} \end{bmatrix}\begin{bmatrix} a \\ \mathsf{I} \end{bmatrix}\begin{bmatrix} \Box \\ \mathsf{I} \end{bmatrix}\begin{bmatrix} \Box \\ \mathsf{I} \end{bmatrix}$$

over the alphabet $\Sigma_{enc^*} = \{a,b,\Box\} \times \{\mathsf{I},\Box\}$. ◁

To decode a valid encoding, we apply two homomorphisms $h_V\colon \Sigma_{enc^*} \to \Sigma^*$ and $h_D\colon \Sigma_{enc^*} \to \Sigma_D^*$ that we define by

$$h_V(a,b) = \begin{cases} a & \text{if } a \in \Sigma; \\ \varepsilon & \text{otherwise;} \end{cases}$$

[2] See Section 2.1 for a definition.

and

$$h_D(a,b) = \begin{cases} \mathsf{I} & \text{if } b = \mathsf{I}; \\ \varepsilon & \text{otherwise}; \end{cases}$$

where $(a,b) \in \Sigma_{enc^*}$. Furthermore, we lift h_V and h_D to words and languages in the usual way. Then, it is not hard to see that $h_V(enc(v,i)) = v$ and $h_D(enc(v,i)) = \mathsf{I}^i$ holds for all $(v,i) \in V \times \mathbb{N}$. In particular, we obtain $\mathrm{Attr}_\sigma(F)$ by applying h_V to $enc^*(X_\sigma)$; that is, $\mathrm{Attr}_\sigma(F) = h_V(enc^*(X_\sigma))$.

An important property in this context, which we use later in this section, is that regular languages are closed under homomorphisms (e.g., see Hopcroft and Ullman [HU79] for a proof). In particular, we exploit the fact that $h_V(enc^*(X_\sigma))$ is a regular language if $enc^*(X_\sigma)$ is.

6.3.2 A Teacher for Automatic Reachability Games

We now describe how to implement a teacher who answers membership and equivalence queries with respect to X_σ. To simplify the description, we first give a generic description without referring to a particular encoding or the fact that we deal with automatic reachability games. At the end of this section, we describe how to construct a teacher based on the encoding enc^* that can perform all necessary operations using finite automata.

Membership queries There are two straightforward ways to answer membership queries of the form "$(v,i) \in X_\sigma$?": one can either perform a forward-search for i steps starting at vertex v and check whether Player σ can force to visit a vertex in F, or one can compute $\mathrm{Attr}^i_\sigma(F)$ and check whether $v \in \mathrm{Attr}^i_\sigma(F)$ holds. Note that storing intermediate results in the latter approach allows the teacher to answer future membership queries whose distance does not exceed i. However, this might come at the price of higher memory consumption.

Equivalence queries On an equivalence query, the teacher is confronted with a set Y and needs to check whether Y equals the unique fixed point X_σ of Γ_σ. To this end, we first check whether $\Gamma_\sigma(Y) = Y$ holds. If this is the case, we return "yes". If this is not the case, we need to identify and return a counterexample (v,i) that satisfies $(v,i) \in X_\sigma$ if and only if $(v,i) \notin Y$.

In order to identify a counterexample, we consider two disjoint cases that can occur if $\Gamma_\sigma(Y) \neq Y$ holds: either $\Gamma_\sigma(Y) \setminus Y \neq \emptyset$ or $\Gamma_\sigma(Y) \subsetneq Y$.

- Let $\Gamma_\sigma(Y) \setminus Y \neq \emptyset$. Moreover, let $(v,i) \in \Gamma_\sigma(Y) \setminus Y$.

If $i = 0$, then we know that $v \in F$ holds by definition of Γ_σ and, hence, $(v,i) \in X_\sigma$. Thus, (v,i) is a counterexample because $(v,i) \in X_\sigma$ and $(v,i) \notin Y$.

If $i > 0$, we ask a membership query with (v,i). If the answer is "yes", then (v,i) is a counterexample since $(v,i) \in X_\sigma$ and $(v,i) \notin Y$. Otherwise, we distinguish whether $v \in V_\sigma$ or $v \in V_{1-\sigma}$.

If $v \in V_\sigma$, then there exists a vertex $v' \in V$ with $(v,v') \in E$ and $(v',i-1) \in Y$ because $(v,i) \in \Gamma_\sigma(Y)$. Moreover, we deduce for every $v' \in V$ with $(v,v') \in E$ that $(v',i-1) \in Y$ implies $(v',i-1) \notin X_\sigma$ (otherwise, $(v,i) \in X_\sigma$ holds). Thus, every $(v',i-1) \in \Gamma_\sigma(Y)$ with $(v,v') \in E$ is a counterexample. Since every vertex of an automatic arena has—by definition—a finite number of outgoing edges, the search for a counterexample is guaranteed to finish in finite time.

If $v \in V_{1-\sigma}$, then we know that $(v',i-1) \in Y$ holds for all vertices $v' \in V$ with $(v,v') \in E$ since $(v,i) \in \Gamma_\sigma(Y)$. In addition, there exists at least one $v' \in V$ such that $(v',i-1) \notin X_\sigma$ (again, $(v,i) \in X_\sigma$ holds otherwise). Thus, every pair $(v',i-1) \notin X_\sigma$ with $(v,v') \in E$ is a counterexample. We find such a pair by asking corresponding membership queries. This search is again guaranteed to terminate because every vertex has a finite number of outgoing edges.

- Let $\Gamma_\sigma(Y) \subsetneq Y$. Then, Y is a so-called *prefixed point*[3]. In addition, by applying Γ_σ to both sides of the inequation $\Gamma_\sigma(Y) \subsetneq Y$ and using the monotonicity of Γ_σ, we get $\Gamma_\sigma(\Gamma_\sigma(Y)) \subsetneq \Gamma_\sigma(Y)$. Hence, $\Gamma_\sigma(Y)$ is also a prefixed point. Finally, the Knaster-Tarski theorem [Tar55] states that the intersection of all prefixed points is a fixed point, which in our case is the set X_σ. Since both Y and $\Gamma_\sigma(Y)$ are prefix points, every pair $(v,i) \in Y \setminus \Gamma_\sigma(Y)$ is not in the intersection of all prefixed points and, therefore, $(v,i) \notin X_\sigma$. This implies that (v,i) is a counterexample.

Answering queries with respect to *enc** Finally, let us show how to realize a teacher who answers membership and equivalence queries with respect to the specific encoding *enc**. Thereby, we assume that the teacher has access to the automata of the automatic reachability game. Furthermore, we assume without loss of generality that a learner never poses queries involving invalid encodings. The learner can ensure this by preceding each query with simple preprocessing steps: in the case of membership queries, the learner independently classifies invalid encoding as "no"; in the case of equivalence queries, the learner intersects a conjecture with a DFA that accepts exactly the set of valid encodings and submits the result to the equivalence query.

[3]Given a set U and a monotonic function $f\colon U \to U$, a *prefix point* is a set $Y \subseteq U$ that satisfies $f(Y) \subseteq Y$.

Answering queries with respect to the encoding enc^* can be done as follows:

- On a membership query, the learner proposes a valid encoding $u \in \Sigma_{enc^*}^*$ and wants to know whether $u \in enc^*(X_\sigma)$ holds. To answer this query, we first decode u using the homomorphisms h_V and h_D: let $(v,i) = (h_V(u), h_D(u))$. Then, we either use an i-step forward-search starting in v, or we compute $\mathrm{Attr}_\sigma^i(F)$ and check whether $v \in \mathrm{Attr}_\sigma^i(F)$ holds. The first approach involves the computation of the sets $R(\mathcal{T})(Y_1), \dots, R(\mathcal{T})(Y_i)$ for a series of finite and, hence, regular sets $Y_1, \dots, Y_i \subseteq V$, which one can do effectively (see Lemma 2.1 on Page 17).[4] For the second approach, one can use Algorithm 6.1 to compute $\mathrm{Attr}_\sigma^i(F)$.

- On an equivalence query, the learner proposes a regular set $Y \subseteq \Sigma_{enc^*}^*$ of valid encodings—we assume that it is given as a DFA—and wants to know whether his conjecture satisfies $Y = enc^*(X_\sigma)$. The key insight as to answering equivalence queries is the fact that we can compute $\Gamma_\sigma(Y)$ analogously to the symbolic attractor computation of Section 6.2 provided that Y is a regular set given as a DFA. A construction similar to the one in the proof of Lemma 6.1 shows that $\Gamma_\sigma(Y)$ is regular if Y is regular, and that one can effectively construct a DFA accepting $\Gamma_\sigma(Y)$ from a DFA accepting Y. The only difference between the symbolic attractor computation of Section 6.2 and the computation of $\Gamma_\sigma(Y)$ lies in the fact that we here also have to take the distance information into account.

  The exact implementation of an equivalence query is tedious and straightforwardly follows the description above. Besides computing $\Gamma_\sigma(Y)$, it involves applying the homomorphisms h_V and h_D, computing Boolean combinations of regular languages, emptiness checks for regular languages, and several more standard operations on regular languages. However, we skip an in-depth description of this construction since it is to a large extent similar to the one in the proof of Lemma 6.1.

6.3.3 Solving Automatic Reachability Games over Infinite Arenas

We can now determine the winning regions and the winning strategies for automatic reachability games over both finite and infinite arenas as follows. Given an automatic reachability game $\mathfrak{G} = (\mathfrak{A}, \mathcal{A}_F)$ over the automatic arena $\mathfrak{A} = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T})$, we construct a teacher for $enc^*(X_0)$, as described in Section 6.3.2, and plug in an active learning algorithm (e.g., Angluin's algorithm). If $enc^*(X_0)$ is a regular language, the learning algorithm eventually terminates and returns a DFA accepting $enc^*(X_0)$. Given this DFA, we compute $\mathrm{Attr}_0(F) = h_V(enc^*(X_0))$, which is again a regular language, and

[4] Let us remind the reader that $R(\mathcal{T})(Y)$ denotes the image of the set Y under the relation defined by the transducer $\mathcal{T}$.

derive the winning regions $W_0 = \mathrm{Attr}_0(F)$ and $W_1 = V \setminus W_0$. Moreover, the set X_0 encodes positional winning strategies for both players:

- A strategy of Player σ is as follows. For a vertex $v \in V_0 \cap (W_0 \setminus F)$, let $i \in \mathbb{N}$ be minimal with the property $(v, i+1) \in X_0$ and $(v, i) \notin X_0$. If a play reaches v, Player σ moves from v to a vertex $v' \in R(\mathcal{T})(\{v\})$ with $(v', i) \in X_0$ (which exists since $v \in \mathrm{Attr}_0^{i+1}(F)$). In order to define a positional winning strategy, we fix the smallest such v' with respect to the canonical order on Σ^*. If $v \in F$ or $v \in W_1$, then Player σ moves to the canonically smallest $v' \in R(\mathcal{T})(\{v\})$.
- A strategy of Player $1-\sigma$ is to stay inside W_1. That is, from a vertex $v \in V_1 \cap W_1$, Player $1-\sigma$ moves to the canonically smallest vertex $v' \in R(\mathcal{T})(\{v\}) \cap W_1$; from any other vertex $v \in V_1 \cap W_0$, Player $1-\sigma$ moves to the canonically smallest vertex $v' \in R(\mathcal{T})(\{v\})$.

It is not hard to verify that both strategies are positional and winning for the respective player. Moreover, given a DFA accepting $enc^*(X_0)$, one can perform all necessary operations to determine the next move of a player by means of standard operations on finite automata. Thus, we obtain the result stated next.

Theorem 6.4. *Let $\mathfrak{G}$ be an automatic reachability game. One can effectively construct a teacher for $enc^*(X_0)$ and obtains the winning regions as well as positional winning strategies for both players by actively learning $enc^*(X_0)$ provided $enc^*(X_0)$ is a regular language.*

Note that $enc^*(X_0)$ is always regular if the underlying arena is finite; therefore, Theorem 6.4 holds for all automatic reachability games over finite arenas. However, the theorem below gives a negative answer to the question of whether the regularity of $enc^*(X_\sigma)$, and $enc^*(X_0)$ in particular, is decidable. The reason for this is that automatic graphs are expressive enough to encode computations of Turing machines.

Theorem 6.5. *The decision problem*

> *"Given an automatic reachability game, an encoding enc, and $\sigma \in \{0, 1\}$. Is $enc(X_\sigma)$ regular?"*

is undecidable.

Proof of Theorem 6.5. We prove Theorem 6.5 by a reduction from the halting problem of deterministic Turing machines to the decision problem of Theorem 6.5. We assume basic familiarity with Turing machines and reductions; if necessary, we refer the reader to Papadimitriou [Pap94] for further details.

Our starting point is the halting problem of deterministic Turing machines, which is the decision problem

"Given a deterministic Turing machine $\mathcal{M}$. Does $\mathcal{M}$ halt when started on the empty tape?".

It is well-know that this decision problem is undecidable.

To prove Theorem 6.5, we construct an automatic reachability game $\mathfrak{G}$ and an encoding *enc* from a given Turing machine $\mathcal{M}$ such that $\mathcal{M}$ halts when started on the empty tape if and only if $enc(X_\sigma)$ is regular. Our construction proceeds in three steps, which we describe next.

First step Given a deterministic Turing machine $\mathcal{M}$, we construct a deterministic Turing machine $\mathcal{M}'$ that works as follows. The machine $\mathcal{M}'$ deletes its input from the tape and simulates $\mathcal{M}$ on the empty tape step-by-step. After the i-th step of this simulation, $\mathcal{M}'$ writes the word $a^i b^i$ at the edge of the tape and immediately deletes it (we assume that neither a nor b are contained in the working alphabet of $\mathcal{M}$). Then, $\mathcal{M}'$ proceeds with the next step of the simulation. Once $\mathcal{M}$ halts, $\mathcal{M}'$ halts, too.

The machine $\mathcal{M}'$ is constructed such that it assumes configurations containing the infix $a^i b^i$ for increasing values of i during the simulation of $\mathcal{M}$. If $\mathcal{M}$ halts on the empty tape, then the set of configurations that $\mathcal{M}'$ assumes during its computation is finite and, therefore, regular. On the other hand, if $\mathcal{M}$ does not halt, $a^i b^i$ occurs for every $i \in \mathbb{N}_+$ as an infix of a configuration of $\mathcal{M}'$, and the set of configurations that $\mathcal{M}'$ assumes is not regular. Thus, the following three statements are equivalent:

- $\mathcal{M}$ halts when started on the empty tape.
- The set of configurations that $\mathcal{M}'$ assumes during its computation is finite.
- The set of configurations that $\mathcal{M}'$ assumes during its computation is regular.

Second step We now construct an automatic reachability game $\mathfrak{G}(\mathcal{M}')$ that depends on the machine $\mathcal{M}'$. The game $\mathfrak{G}(\mathcal{M}') = (\mathfrak{A}, F)$ is a solitary game (i.e., only Player σ plays) whose arena is the configuration graph of $\mathcal{M}'$ with reversed edges. More precisely, the set V_σ is the set of all configurations of $\mathcal{M}'$, the set $V_{1-\sigma}$ is empty, and there exists an edge from configuration c to configuration c' if $\mathcal{M}'$ can move from c' to c. Moreover, $F = \{c_0\}$ is a singleton set containing the start configuration c_0 in which $\mathcal{M}'$ starts in its initial state on the empty tape. We refer the reader to Thomas [Tho01] for details about how to represent the configuration graph of a Turing machine as an automatic graph.

It is not hard to verify that $\mathrm{Attr}_\sigma(F)$ of the game $\mathfrak{G}(\mathcal{M}')$ coincides with the set of configurations that $\mathcal{M}'$ assumes when started on the empty tape. Thus, the following statements are equivalent:

- The set of configurations that $\mathcal{M}'$ assumes during its computation is regular.
- The attractor $\mathrm{Attr}_\sigma(F)$ of the game $\mathfrak{G}(\mathcal{M}')$ is finite.
- The attractor $\mathrm{Attr}_\sigma(F)$ of the game $\mathfrak{G}(\mathcal{M}')$ is regular.

Third step We fix the encoding to be enc^* from Section 6.3.1. Moreover, let X_σ be the fixed point of Γ_σ with respect to the game $\mathfrak{G}(\mathcal{M}')$. Then, we claim that the following statements are equivalent:

1. The attractor $\mathrm{Attr}_\sigma(F)$ of the game $\mathfrak{G}(\mathcal{M}')$ is regular.
2. The set $enc^*(X_\sigma)$ is regular.

Let us first prove that Statement 1 implies Statement 2. To this end, let $\mathrm{Attr}_\sigma(F)$ be regular. Then, we know that $\mathrm{Attr}_\sigma(F)$ is finite. Moreover, if $\mathrm{Attr}_\sigma(F)$ is finite, then X_σ has a special form in that the set $\{v \in V \mid \exists i \in \mathbb{N} \colon (v, i) \in X_\sigma\}$ is finite, too. In this case, $enc^*(X_\sigma)$ is regular.

To prove the reverse direction, we apply a well-known fact from formal language theory, namely that the image $h(L)$ of a regular language $L \subseteq \Sigma_1^*$ under a homomorphism $h \colon \Sigma_1 \to \Sigma_2^*$ is again a regular language (e.g., see Hopcroft and Ullman [HU79]). Thus, we immediately obtain that $h_V(enc^*(X_\sigma)) = \mathrm{Attr}_\sigma(F)$ is regular if $enc^*(X_\sigma)$ is regular.

To conclude, the Turing machine $\mathcal{M}$ halts when started on the empty tape if and only if the set $enc^*(X_\sigma)$ is regular. Thus, the decision problem of Theorem 6.5 is undecidable. □

Let us conclude this section with a remark about the time and space complexity of our learning-based algorithm. The time and space used by our approach depends on the number of queries the chosen learning algorithm asks. This number is polynomial in the size n of the minimal DFA accepting $enc^*(X_\sigma)$ when using Angluin's or Kearns and Vazirani's algorithm. More precisely, let d be a bound for the number of outgoing edges per vertex and t the size of the transducer $\mathcal{T}$; then, answering an equivalence query requires the teacher to construct at most d automata whose size can be bounded by $2^{\mathcal{O}(n \cdot t)}$. The length of a counterexample can also be bounded by $2^{\mathcal{O}(n \cdot t)}$, and the teacher conducts at most $d+1$ membership queries during any equivalence query. To answer a membership query with a pair (v, i), the teacher either computes $\mathrm{Attr}_0^i(F)$ or performs a forward-search in the game graph. The first approach produces a sequence of DFAs whose size might grow exponentially, resulting in a nonelementary complexity. On the other hand, the teacher can perform the search of the second approach in at most $d^{2^{\mathcal{O}(n \cdot t)}}$ steps. Provided the teacher uses the forward-search as described above, we obtain a doubly-exponential algorithm.

6.4 Experiments and Evaluation

In this section, we evaluate the algorithms developed in Sections 6.2 and 6.3 based on a prototype implementation. This evaluation builds upon the work by Alur, Madhusudan, and Nam [AMN05b], in which the authors provide reachability games and compare several symbolic techniques for solving these games.

We describe the games used by Alur, Madhusudan, and Nam in Section 6.4.1. Since these games are not formulated in a form that allows applying our methods directly, we also provide a translation into automatic reachability games. Subsequently, in Section 6.4.2, we present and discuss experimental results of our prototype on these games.

6.4.1 A Framework to Evaluate Symbolic Techniques for Reachability Games

Alur, Madhusudan, and Nam [AMN05b] have introduced a common framework to compare and evaluate different symbolic techniques for solving reachability games. This framework consists of two games, the *pursuit-evasion game* and the *swap game*, which both can be scaled by a parameter $n \in \mathbb{N}_+$. However, these games do not comply with our definition of automatic reachability games: for one thing, the games involve concurrent interaction of the players, which is not part of our model and needs to be eliminated; for another, since these games are not formulated as *automatic* reachability games, we have to provide a translation into a suitable representation in terms of regular languages. We describe our solutions to both tasks after introducing the games.

The games The *pursuit-evasion game*, sketched in Figure 6.3 (on Page 198), is a game between a pursuer P and an evader E played on an $n \times n$ grid. The pursuer's objective is to catch the evader by occupying the same field as the evader before the latter can reach the top-left position (this position is shaded gray in Figure 6.3). Conversely, the evader's objective is to reach the top-left position without being caught by the pursuer. The players move concurrently; in each turn, they can either stay or move up, down, left, or right. The pursuer, however, is only allowed to move in every even turn and has to stay stationary in every odd turn. We refer the reader to Alur, Madhusudan, and Nam [AMN05b] for a comprehensive definition of the pursuit-evasion game.

The second game is the *swap game*. It originates from the "swap example", which has been introduced by McMillan [McM02] and was subsequently turned into a game [AMN05b]. We here consider a slight variant, to which we also refer to as swap game for the sake of simplicity. In this game, an array a of length n with entries $a[i] \in [n]$ is manipulated by two players, the *system* and the *environment*. Starting with

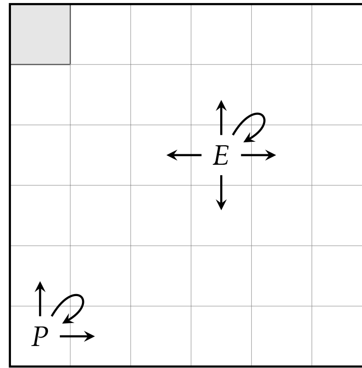


Figure 6.3: The pursuit-evasion game. E represents the evader and P the pursuer. The evader's safe area is shaded gray. The depiction is taken from [AMN05b].

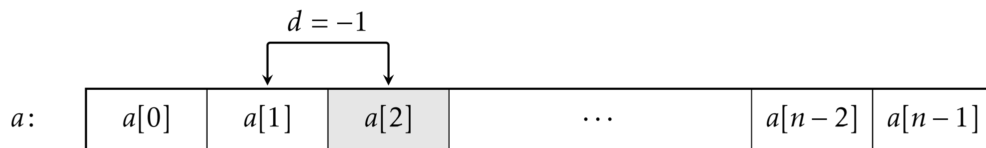


Figure 6.4: The swap game. In the sketched situation, the system chose the direction $d = -1$ while the environment chose the index $i = 2$ (the corresponding array entry is shaded gray).

an arbitrarily initialized array, each round of the game proceeds as follows: the system chooses a direction $d \in \{-1, 1\}$, meaning either left or right, while the environment chooses an index $i \in [n]$; then, the array entries $a[i]$ and $a[(i + d) \bmod n]$ are swapped. The objective of the system is to eventually satisfy $a[0] = a[1]$, whereas the environment wants to prevent this. Figure 6.4 illustrates the swap game.

The pursuit-evasion game and the swap game both comprise a finite number of *states*[5], and the translation into an automatic reachability game (as described later) yields a game over a finite arena. To evaluate our learning-based algorithm also on automatic reachability games over infinite arenas, we additionally design an unbounded version of the swap game, which we call the *unbounded swap game*. The unbounded swap game is the same game as the original swap game, except that the length of the array is no longer fixed a priori. That means, the array entries still satisfy $a[i] \in [n]$, but the game can start with an array of arbitrary but finite length. In this way, we obtain a game with infinitely many states.

Removing concurrency The games described above involve concurrent interaction between the players whereas our definition of reachability games requires the players

[5] A *state* refers to a complete description of the current situation in a game (i.e., the players' positions on the grid, or the content of arrays, and whose turn it is).

to move successively. In these so-called *concurrent reachability games* (cf. de Alfaro, Henzinger, and Kupferman [dAHK07]) the players typically play randomized strategies, which win with a certain probability. However, if one is interested in strategies that win every play (not just with probability 1), then one can reformulate this type of games to match our definition of reachability games. The idea is to let Player 0 move first and allow the opposing player to make his move subsequently, depending on this additional knowledge. Clearly, if Player 0 has a winning strategy in the reformulated game, then this strategy is also a winning strategy in the concurrent game.

In the following, we use this reformulated version.

Translation into automatic reachability games In order to translate the games described above into automatic reachability games, we represent the states of a game as finite words. We consider three ways to do so, each using a different representation of the *values* that constitute a state (i.e., the positions on the grid or the entries of an array). The first representation encodes values in unary, the second in binary, and the third encodes each value by a distinct symbol of the alphabet. The choice of the representation has a decisive influence on the size of the resulting DFAs.

Unary representation The unary representation uses the alphabet $\Sigma_u = \{0,1\}$ and encodes the value $i \in [n]$ as the word $1^i 0^{n-i-1}$. In the case of the pursuit-evasion game, the unary representation encodes the players' positions on the grid as the word $u_x u_y v_x v_y$ where the infixes u_x, v_x are the unary encodings of the players' x-positions and the infixes u_y, v_y are the unary encodings of the players' y-positions. In the case of the swap game and the unbounded swap game, the unary representation encodes an array of length n as the word $u_0 \dots u_{n-1}$ where each infix u_i is the unary encoding of the i-th array entry.

Binary representation The binary representation works in the same way as the unary representation, except that it encodes values from $[n]$ as words over $\Sigma_b = \{0,1\}$ in an m-bit binary encoding with $m = \lceil \log_2 n \rceil$.

Alphabet representation The alphabet representation works in the same way as both representations above, except that it encodes the value $i \in [n]$ as the symbol i itself. That means, the alphabet used by the alphabet representation is $\Sigma_a = [n]$.

A complete description of the state of a game also has to include further information such as whose turn it is. We add this information by prefixing the representations from above with an appropriate word.

Each of the above representations encodes a state of a game as a unique word, and the set of all such words constitutes the vertex set of the resulting game. It is not hard

to verify that each representation indeed yields an automatic reachability game; that is, one can construct DFAs accepting the sets V_0, V_1, and F as well as a synchronous transducer defining the game's edge relation.[6] Note that the pursuit-evasion game and the swap game both translate into automatic reachability games over finite arenas whereas the unbounded swap game translates into one over an infinite arena.

Our motivation for using distinct representations of the states of a game is to study the performance of our algorithm when confronted with different DFAs (of different sizes and alphabets) representing the same game. In particular, we want to find out whether there exists a trade-off between the number of states and the alphabet size of the DFAs representing an automatic reachability game. Note that the alphabet size of the unary and binary representation is fixed, whereas it grows with the parameter n in the alphabet representation.

6.4.2 Experiments

To assess the techniques described in Sections 6.2 and 6.3, we implemented a prototype and evaluated its performance on the games described above.

Methodology We implemented our prototype in Java. To perform operations on automata, we used the `dk.brics.automaton` finite automata library [Mø10]. The prototype is connected to Libalf via the Java Native Interface, but it also features a native Java implementation of Kearns and Vazirani's algorithm.

We created nine different benchmark suites, one for each combination of game and representation. Each of these benchmark suites consists of several automatic reachability games, which we obtained by varying the parameter n. Tables 6.2a, 6.2b, and 6.3 present further details.

We conducted the experiments on an Intel Q9550 quad core CPU at 2.83 GHz with 4 GiB of RAM running Ubuntu 12.04 LTS. We used a 10 h timeout limit, which was also the time limit used by Alur, Madhusudan, and Nam. For reasons of efficiency, we chose the native Java implementation of Kearns and Vazirani's algorithm (as it turned out that data exchange rate between the Java and the C++ part was very low). The prototype uses the DFA-based attractor computation of Section 6.2 to answer membership queries (although we also implemented an explicit forward-search).

Results Tables 6.2a, 6.2b, and 6.3 present the experimental results. Tables 6.2a and 6.2b show the results of the DFA-based attractor computation of Section 6.2 on the pursuit-evasion game and the swap game, respectively. Table 6.3 shows the results of the learning algorithm of Section 6.3 on the unbounded swap game. The columns

[6] A complete description of these translations is simple but lengthy. Thus, we skip further details here.

Table 6.1: Experimental results of the DFA-based attractor computation on games over finite arenas.

(a) Results on the pursuit-evasion game.

Repr.	n	Steps	States	Runtime in s
Unary	4	10	410	1.3
	8	22	2614	5.6
	16	46	18 286	163.9
	32	94	131 486	28 351.4
	64	—	—	—
Binary	4	10	117	0.6
	8	22	726	1.8
	16	46	3895	17.6
	32	94	17 624	338.8
	64	190	75 283	15 764.4
Alph.	4	10	62	0.6
	8	22	250	1.3
	16	46	1004	13.6
	32	94	3868	301.0
	64	—	—	—

(b) Results on the swap game.

Repr.	n	Steps	States	Runtime in s
Unary	4	4	159	0.8
	5	6	452	3.3
	6	8	1173	129.4
	7	10	2881	26 702.8
	8	—	—	—
Binary	4	4	335	0.9
	5	6	708	5.5
	6	8	1077	103.6
	7	10	1288	4218.3
	8	—	—	—
Alph.	4	4	30	0.3
	5	6	62	1.0
	6	8	126	7.3
	7	10	254	125.8
	8	12	510	12 317.0

"Repr." refer to the representations used to encode the games ("Alph." means alphabet encoding), and the columns "n" show the parameter used to vary the size of the game. The columns "Size" display the number of states of the final DFAs, and the columns "Runtime" show the runtime of the prototype in seconds. The columns "Steps" of Tables 6.2a and 6.2b report the number of iterations until the attractor computation became stationary. The columns "EQ" and "MQ" of Table 6.3 report the number of membership queries and equivalence queries, respectively. A "—" entry means that either Java ran out of memory or the computation did not finish within the time limit.

Results on games over finite arenas In the case of the pursuit-evasion game, the prototype solved the game in the unary and the alphabet representation up to a grid of size 32×32 ($n = 32$) and in the binary representation up to a grid of size 64×64 ($n = 64$). In the case of the swap game, the prototype solved the game in the unary and the binary representation up to arrays of length $n = 7$ and in the alphabet representation up to arrays of length $n = 8$.

Applying our learning-based technique to games over finite arenas turned out to be less successful than the DFA-based attractor computation (which is why we do not present details here). In the case of the pursuit-evasion game, the prototype achieved the best result for the alphabet representation and solved the game up to $n = 16$.

Table 6.3: Experimental results of the learning-based algorithm on the unbounded swap game.

Repr.	n	States	Membership queries	Equivalence queries	Runtime in sec.
Unary	5	33	1126	33	27.1
	6	37	1454	37	81.6
	7	49	2466	49	1153.0
	8	53	3021	53	13710.5
	9	—	—	—	—
Binary	2	11	145	11	1.0
	3	13	182	13	2.5
	4	17	320	17	18.6
	5	21	518	21	505.3
	6	—	—	—	—
Alph.	5	5	74	5	0.2
	6	5	84	5	0.2
	7	5	94	5	0.2
	8	5	104	5	0.2
	250	5	2524	5	4.7

In the case of the swap game, the prototype achieved the best result for the unary representation and solved the game up to an array of length $n = 6$.

Experimental results of games over infinite arenas The prototype solved the unbounded swap game in the unary representation up to $n = 8$ and in the binary representation up to $n = 6$. In the case of the alphabet representation, the prototype performed considerably better and solved the game up to $n = 250$ (we did not conduct experiments for any larger value).

As shown in Table 6.3, the number of membership and equivalence queries was moderate, up to 3021 membership queries and 53 equivalence queries. Moreover, the prototype spent almost the entire time answering membership queries (which, however, is not shown in Table 6.3.)

Discussion To assess the performance of our DFA-based attractor computation on games over finite arenas, we contrast the results of our prototype to a benchmark by Alur, Madhusudan, and Nam [AMN05b]. Subsequently, we discuss the results on games over infinite arenas.

Finite arenas Alur, Madhusudan, and Nam compared three symbolic algorithms on the pursuit-evasion game and the swap game:

Table 6.4: Summary of the benchmark by Alur, Madhusudan, and Nam [AMN05b] and our results on reachability games over finite arenas.

	Maximal solvable game size n			
Game	Automatic	BDD	QBF	SAT
Pursuit-evasion game	64	32 (512)	8	32
Swap game	8	9	8	9

- Symbolic fixed-point computations using *binary decision diagrams (BDDs)*, one version based on the Mocha tool [AHM+98] and a second version based on the μcke tool [Bie97]
- Reductions to *Propositional Boolean Logic (SAT)* using several SAT solvers
- Reductions to *quantified Boolean formulas (QBF)* using several QBF solvers

In order to reduce a reachability game to Propositional Boolean Logic and quantified Boolean formulas, Alur, Madhusudan, and Nam considered a bounded version, in which one is interested in reaching a target set within a fixed number of moves.[7]

Table 6.4 summarizes the benchmark by Alur, Madhusudan, and Nam by highlighting the best results of each considered technique. As reference, the table also includes the best results of our DFA-based attractor computation, which is shown in the column "Automatic".

The experimental results of our DFA-based attractor computation are similar to the results of Alur, Madhusudan, and Nam [AMN05b]. Only one algorithm based on the μcke tool performed considerably better on the pursuit-evasion game and was able to solve the game for a grid of size 512×512. The QBF-based method performed worse than our prototype on the pursuit-evasion game but matched its performance on the swap game.

The main result of our experiments on games over finite arenas is that our DFA-based attractor computation is competitive to the state-of-the-art methods studied by Alur, Madhusudan, and Nam. However, different representations of games seemed to have no considerable influence on the maximal tractable size of the game, and no representation was clearly superior.

To conclude the discussion about games over finite arenas, let us comment on the observation that the learning-based technique was less effective than the DFA-based attractor computation on games over finite arenas. It turned out that the reason for this behavior was the following: the learner posed membership queries with pairs

[7] We refer the reader to Alur, Madhusudan, and Nam [AMN05b] for a comprehensive description.

(v, i) where i was large enough to make the teacher compute the complete attractor in order to answer the query. Even falling back to the explicit forward-search described in Section 6.3.2 did not improve the performance because the search almost always ran out of memory. Note, however, that the learner's behavior depends on the chosen representation. Thus, the performance of our prototype on the pursuit-evasion game and the swap game is not a general drawback of our learning-based method but due to the considered representations.

Infinite arenas The main result of our experiments on games over infinite arenas is that our prototype was able to solve large instances of the unbounded swap game. The prototype performed particularly well on the alphabet representation. In summary, the experiments on the unbounded swap game demonstrate that our learning-based method is a promising technique for solving reachability games over infinite arenas.

However, the experiments also showed that answering membership queries is a bottleneck. Thus, one should employ a learning algorithm that poses as few membership queries as possible. Moreover, the experiments for the unbounded swap game show that the representation plays an important role and that its choice needs to be made carefully.

6.5 Conclusion

In this chapter, we studied automatic reachability games, which are reachability games played on arenas whose underlying graphs are automatic. We considered two different settings: automatic reachability games over finite arenas and such over infinite arenas.

To solve automatic reachability games over finite arenas, we presented a symbolic fixed-point algorithm, which is based on an iterative computation of the attractor by means of DFAs. To solve automatic reachability games over infinite arenas, we have developed a learning-based algorithm, which works in Angluin's original active learning setting. We demonstrated the performance of our techniques, using a prototype implementation, on several reachability games. Our experiments show that the DFA-based fixed-point algorithm is competitive to other symbolic techniques for solving reachability games over finite arenas, such as BDD-based methods or translations into logic formulas. In addition, the learning-based method proved to be successful for solving reachability games over infinite arenas.

A natural extension of automatic reachability games are rational reachability games (i.e., reachability games played on arenas whose underlying graph is rational). In fact, the techniques developed in this chapter are not limited to automatic reachability games but also work for rational reachability games; more precisely, both Theorems 6.2

and 6.4 also hold if the arena is rational, represented by an asynchronous transducer. The reason for this is that neither the DFA-based attractor computation nor the learning-based technique relies on properties that are exclusive to automatic graphs, respectively synchronous transducers. On the contrary, all necessary operations on the underlying transducers, such as computing predecessor vertices, can be performed on asynchronous transducers as well, and both of our algorithms can in fact be applied to rational reachability games without any changes.

A promising direction of future research is to extend this work to more expressive language classes, such as languages recognizable by realtime or visibly one-counter automata. Both language classes enjoy good closure properties and Angluin-style active algorithms are available [FR95, NL10].

Moreover, it would be interesting to lift our learning-based technique to more complex winning conditions such as Büchi, parity, or Muller winning conditions. The work of Vardhan et al. [VSVA05], in which the authors describe how to verify ω-regular properties by learning appropriate fixed points, makes us convinced that this is indeed a worthwhile direction to proceed.

7

Labeled Safety Games

Our final application scenario for automata learning is *labeled safety games*. In contrast to the classical setting, as described in Section 2.3, the arena's edges are now *deterministically labeled with actions*. Games of this type are played similarly to classical safety games: two players move a token from one vertex to the next, and the objective of Player 0 is to stay inside a designated set of "safe" vertices; this time, however, a player's move is to pick an action—as opposed to choosing a successor vertex—, and a play is a sequence of actions rather than a sequence of vertices. This entails that a strategy now has to operate on a sequence of actions and cannot rely on the history of visited vertices as external input.

Consequently, a controller (i.e., a physical implementation of a strategy such as a circuit or a piece of software) needs to track the vertices occurring during a play in order to determine the action to play next. In the terminology of Ehlers [Ehl11], such implementations are called *stand-alone strategies* because they are independent of the game's arena. However, in order to track the vertices visited so far, a controller needs memory of some kind. We here follow a common approach in the literature (see Grädel, Thomas, and Wilke [GTW02] for an overview) and implement a strategy in terms of a finite-state machine that uses its states as memory. This view on strategies allows us to define the *size* of the implementation of a strategy as the number of states of the underlying finite-state machine.

When infinite games are used for controller synthesis, the actual size of the implementation of a strategy is often a crucial factor in applications (e.g., as argued by Bloem et al. [BGJ+07]). This gives rise to the question of how to synthesize winning strategies that result in small or even minimal implementations. Although much research on efficient algorithms to synthesize winning strategies for various types of infinite games has been spend during the last decades (again, see Grädel, Thomas, and

Wilke [GTW02]), the question of how to synthesize small implementations has been much less studied; this is in particular true for safety games. In fact, this question seems to be hard to settle, and there is no satisfactory answer till today (although some progress have been made, which we discuss shortly).

A first approach to synthesizing strategies for labeled safety games is applying the standard attractor computation and lifting the resulting positional strategies to the present setting. The common way to achieve the latter is to use the part of the arena as memory structure that is reachable by the player's strategy and the different choices of the opposing player (e.g., this approach is used by Ehlers [Ehl11]). The result is an implementation in form of a finite-state machine that tracks the vertices of a play and uniquely determines the next action depending on the current vertex.

Although existing strategy synthesis tools, such as Gavs+ [CKLB11], implement efficient versions of the attractor computation, they do not take the size of the resulting implementation into account. Instead, these tools construct positional strategies by means of purely greedy approaches, which cannot make any guarantees about the size of the resulting implementation. However, it turns out that we cannot hope for a polynomial time algorithm to find minimal implementations of strategies: we show that the corresponding decision problem is NP-complete.

We approach the problem of synthesizing strategies for labeled safety games by devising a learning-based heuristic, which learns winning strategies whose implementations are often small. Our algorithm is based on active learning in an Angluin setting and runs in time polynomial in the size of the game's arena. The idea of our approach is to consider finite sequences of actions, which we call *finite traces*, and to define implementations of strategies in terms of DFAs, which we call *strategy automata*. Synthesizing winning strategies with small implementations then amounts to synthesizing winning strategy automata with few states. We set up all necessary formalisms in Section 7.1.

Since one can solve labeled safety games efficiently (e.g., by computing a positional strategy using a tool such as Gavs+), we can easily determine which finite trace belongs to a winning play and which does not. Based on this knowledge, we construct a teacher, pit him against an active learning algorithm (e.g., Angluin's algorithm), and prematurely stop the learning as soon as the learner conjectures a winning strategy automaton. Thereby, we exploit two common properties of active learning algorithms: first, active learning algorithms typically produce conjectures of monotonically increasing size (which implies that small automata are conjectured first); second, active learning algorithms are good at generalizing partial knowledge (which helps finding a winning strategy automaton early in the learning process). In fact, the combination of both properties often allows us to stop the learning early, which then results in a small winning strategy automaton. We explain this procedure thoroughly in Section 7.2.

It turns out that the classical active learning algorithms, such as Angluin's algorithm or Kearns and Vazirani's algorithm, tend to learn unnecessary large strategy automata. The intuitive reason for this behavior is the way these algorithm explore transitions: once the algorithm has discovered a new state, it explores all outgoing transitions of this state. In the context of learning strategies, however, this refers to exploring all possible ways to continue playing, which is clearly counterproductive towards the goal of learning a small strategy automaton (as one way to continue playing suffices). Therefore, we develop variations of Angluin's as well as Kearns and Vazirani's algorithm, which avoid this undesired behavior by exploring transitions only when necessary. We present these variations in Section 7.2.2.

Finally, we demonstrate the effectiveness of our heuristic through a series of experiments with a prototype implementation. Section 7.3 presents the results of these experiments, which substantiate that learning indeed helps synthesizing implementations of strategies that are smaller than those arising from positional strategies. A similar comparison, though based on the notion of *sparse strategies*, was recently conducted by Ehlers and Moldovan [EM12], where the authors benchmarked techniques based on integer linear programming, SAT encodings, and our technique. Ehlers and Moldovan found that our technique performs particularly well whenever the strategy implemented by a strategy automation is not positional.

The results of this chapter partly appeared in conference proceedings [Nei11].

Related Work

The question of the "size of a strategy implementation" has most commonly been addressed in the context of Muller games. Strategies for Muller games in general require memory, which is usually realized in terms of Mealy machines that read finite play prefixes and output the players' next moves. It is well known that this approach entails substantial complexity challenges because the resulting Mealy machines are in general exponential in the size of the underlying arena [HD05, DJW97].

Despite these challenges, some approaches to find compact implementations of strategies have been proposed. For instance, Bloem et al. [BGJ+07] considered synthesizing small circuits from specifications written in the Property Specification Language. Another approach (e.g., followed by Holtmann and Löding [HL07] as well as Gelderie and Holtmann [GH11]) is more general in that it reduces the size of a Mealy machine once it has been synthesized. This approach, however, cannot reduce the size of Mealy machines for positional strategies because strategies of this kind already translate to single-state Mealy machines. Thus, it seems unsuited to safety games, for which one typically computes positional winning strategies.

Recently, Gelderie [Gel12] proposed *strategy machines*, which are a particular type of Turing machines, as an alternative means for implementing strategies. Besides the size of a strategy machine, this model allows investigating several other properties, such as the number of steps needed to transform the machine's input into an output, the amount of information that needs to be stored on the tape, and so on. For the properties mentioned above, Gelderie gives lower bounds for Muller, Streett, and LTL games, but he does not explicitly consider safety games.

Relatively little research has been devoted to the special case of finding small implementations of strategies for safety games. Typically, safety games are solved by means of the attractor computation, whose output is used to derive positional winning strategies. Although any positional strategy is optimal in the case of classical safety games (in that it translates to a one-state Mealy machine), this is no longer true for labeled safety games. In fact, Ehlers [Ehl11] showed that finding a stand-alone strategy of minimal size is NP-complete. In a subsequent study, Ehlers and Moldovan [EM12] investigated several methods, including the one presented in this chapter, to find *sparse* strategies for safety games; in this context, "sparse" refers to the number of choices a player has whenever it is his turn. To the best of our knowledge, the work of Ehlers [Ehl11] and the work of Neider [Nei11] are the only ones that explicitly study strategies for safety games in the present context.

7.1 Labeled Safety Games and Strategy Automata

We first set up labeled safety games and adapt the notion of strategies to this new setting (Section 7.1.1). Then, we introduce *strategy automata* (Section 7.1.2). We move on to defining the size of strategy implementations (Section 7.1.3) and, finally, show that computing minimal strategy automata is hard (Section 7.1.4).

7.1.1 Labeled Safety Games

We consider safety games that are played on *finite arenas* whose edges are deterministically labeled with *actions* taken from an alphabet Σ. We call such arenas Σ-*arenas* to distinguish them from those of previous chapters.

Definition 7.1 (Σ-arena)**.** Let Σ be an alphabet. A Σ-*arena* is a tuple $\mathfrak{A} = (V_0, V_1, E)$ where $V = V_1 \uplus V_0$ is a finite set of vertices and $E \subseteq V \times \Sigma \times V$ is a directed, deterministic Σ-labeled edge relation (i.e., $(v, a, v') \in E$ and $(v, a, v'') \in E$ implies $v' = v''$).

As in previous chapters, we assume without loss of generality that all vertices have at least one outgoing edge. Note that Σ-arenas are no restriction: one can convert any

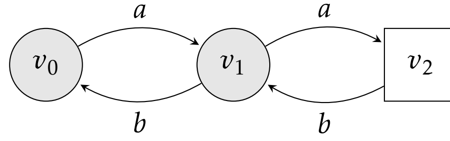


Figure 7.1: An example of a labeled safety game over an $\{a,b\}$-arena. Vertices belonging to F are shaded gray.

finite arena $\mathfrak{A} = (V_0, V_1, E)$ as defined in Section 2.3 into the V-arena $\mathfrak{A}' = (V_0, V_1, E')$ with $E' = \{(v, v', v') \mid (v, v') \in E\}$.

The safety games of this chapter, which we call Σ-*labeled safety games*, consist of a Σ-arena and a set of *safe vertices*.

Definition 7.2 (Σ-labeled safety game)**.** Let Σ be an alphabet. A Σ-*labeled safety game* is a pair $\mathfrak{G} = (\mathfrak{A}, F)$ consisting of a Σ-arena $\mathfrak{A} = (V_0, V_1, E)$ and a set $F \subseteq V_0 \uplus V_1$ of *safe vertices*.

We usually omit Σ if it is clear from context. Moreover, we refer to the safety games defined in Section 2.3 as *classical safety games* to distinguish them from labeled safety games.

Example 7.1. Figure 7.1 depicts an example of a labeled safety game over an $\{a,b\}$-arena with $F = \{v_0, v_1\}$. Note that a transition does not need to be defined for every pair of vertex and action; for instance, this is the case for the vertices v_0 and v_2. ◃

A labeled safety game over a Σ-arena $\mathfrak{A} = (V_0, V_1, E)$ is played by two players, Player 0 and Player 1, as follows: a token is placed on some *initial vertex* v_0 and, depending on whether $v_0 \in V_0$ or $v_0 \in V_1$, the corresponding player chooses an action $a_0 \in \Sigma$ such that an edge $(v_0, a_0, v_1) \in E$ exists; then, he moves the token from vertex v_0 to v_1. Both players repeat this process of choosing actions and moving the token ad infinitum, thereby producing an infinite sequence of actions, which we call a *trace*. Formally, a *trace* is an infinite word $\tau = a_0 a_1 \ldots \in \Sigma^\omega$ for which an infinite sequence of vertices $v_0 v_1 \ldots \in V^\omega$ exists that satisfies $(v_i, a_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$. Note that a trace itself does not carry the information in which vertex it starts, which is why we use the notation τ_{v_0} to indicate that the trace τ starts in v_0—we omit the subscript v_0 if the initial vertex is clear from the context. Analogous to the case of finite plays, we call a finite prefix $u \in \Sigma^*$ of a trace a *finite trace*.

Since Σ-arenas are deterministic, each trace $\tau_{v_0} = a_0 a_1 \ldots \in \Sigma^\omega$ induces a unique play $\rho(\tau_{v_0}) = v_0 v_1 \ldots \in V^\omega$ for which $(v_i, a_i, v_{i+1}) \in E$ is satisfied for all $i \in \mathbb{N}$. However, the converse is not true: there may be distinct traces (even starting in the same initial vertex) that induce the same play. As a shorthand-notation, we write $\mathfrak{A} \colon v_0 \xrightarrow{u} v$ if the

finite trace $u = a_0 \dots a_n \in \Sigma^*$ induces the finite play prefix $\rho(u) = v_0 \dots v_{n+1}$ in the arena $\mathfrak{A}$ and $v_{n+1} = v$; in particular, $\mathfrak{A}\colon v_0 \xrightarrow{\varepsilon} v_0$ holds.

A trace τ_{v_0} is *winning for Player 0* if the induced play $\rho(\tau_{v_0}) = v_0 v_1 \dots$ satisfies $v_i \in F$ for all $i \in \mathbb{N}$. If a trace is not winning for Player 0, then it is winning for Player 1. Moreover, we define that a player loses if he picks an action in one of his vertices for which no outgoing edge exists.

In the present setting, the players make the choice of their next move depending on the trace played so far. To capture this semantic, we adapt the notion of strategies and introduce *trace strategies*.

Definition 7.3 (Trace strategy)**.** Let $\mathfrak{G} = (\mathfrak{A}, F)$ be a labeled safety game over the Σ-arena $\mathfrak{A} = (V_0, V_1, E)$, and let $V = V_0 \uplus V_1$. A *trace strategy* for Player σ, $\sigma \in \{0, 1\}$, in $\mathfrak{G}$ from a vertex $v_0 \in V$ is a partial mapping $f_\sigma\colon \Sigma^* \dashrightarrow \Sigma$ that maps every finite trace $u \in \Sigma^*$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$ and $v \in V_\sigma$ to an action $a \in \Sigma$ such that a vertex $v' \in V$ with $(v, a, v') \in E$ exists.

Given a trace strategy f_σ for Player σ, a trace $\tau_{v_0} = a_0 a_1 \dots$ is *played according f_σ* if $a_{n+1} = f_\sigma(a_0 \dots a_n)$ holds for all $n \in \mathbb{N}$ with $\rho(a_0 \dots a_n) = v_0 \dots v_{n+1}$ and $v_{n+1} \in V_\sigma$. A trace strategy f is *winning for Player σ from vertex v_0* if all traces starting in v_0 and played according to f are winning for Player σ. Winning regions and determinacy are analogous to classical safety games (see Section 2.3).

Example 7.2. Let us reconsider the labeled safety game of Figure 7.1 (on Page 211). The trace $\tau_{v_0} = (ab)^\omega$ is winning for Player 0 from vertex v_0. Moreover,

$$f(u) = \begin{cases} a & \text{if } |u| \text{ is even;} \\ b & \text{if } |u| \text{ is odd;} \end{cases}$$

where $u \in \{a, b\}$ is a trace strategy that is winning for Player 0 from the vertex v_0 (but not from the vertices v_1 or v_2). ◁

Labeled safety games and classical safety games are closely related. In fact, one can view a labeled safety game as a classical safety game by just forgetting about the actions in order to compute winning regions and winning strategies. Let us make this intuition precise.

Definition 7.4. Given a Σ-labeled safety game $\mathfrak{G} = (\mathfrak{A}, F)$ over the Σ-arena $\mathfrak{A} = (V_0, V_1, E)$, the corresponding classical safety game is the game $\mathfrak{G}_\perp = (\mathfrak{A}_\perp, F)$ over the arena $\mathfrak{A}_\perp = (V_0, V_1, E_\perp)$ with $E_\perp = \{(v, v') \mid \exists a \in \Sigma\colon (v, a, v') \in E\}$.

Definition 7.4 allows us to establish a connection between trace strategies for labeled safety games and strategies for classical safety games.

Lemma 7.1. *Let $\mathfrak{G} = (\mathfrak{A}, F)$ be a labeled safety game and $\mathfrak{G}_\perp$ the corresponding classical safety game. Moreover, let $\sigma \in \{0,1\}$, $f_\sigma \colon V^* V_\sigma \to V$ be a strategy for Player σ in $\mathfrak{G}_\perp$, and $v \in V_0 \uplus V_1$ a vertex. Then, the trace strategy*

$$f'_\sigma(u) = \begin{cases} a & \text{if } \rho(u) = v \dots v_n,\ v_n \in V_\sigma,\ f_\sigma(\rho(u)) = v', \text{ and } (v_n, a, v') \in E; \\ \text{undefined} & \text{otherwise}; \end{cases}$$

is winning for Player σ in $\mathfrak{G}$ from vertex v if and only if f_σ is winning for Player σ in $\mathfrak{G}_\perp$ from vertex v.

A standard induction over the length of plays is enough to prove Lemma 7.1; to prove the direction from left to right, one exploits that Σ-arenas are deterministic, which implies that every trace induces a unique play.

Given the fact that winning strategies can be translated into winning trace strategies and vice versa, we immediately obtain that labeled safety games are determined. In particular, the winning regions of a labeled safety game $\mathfrak{G}$ and the corresponding classical safety game $\mathfrak{G}_\perp$ coincide and both can be computed using the classical attractor computation.

Since computing winning strategies for Player 1 in (labeled) safety games amounts to solving reachability games, we restrict ourselves to computing winning strategies for Player 0. Moreover, since trace strategies are defined with respect to an initial vertex, we fix an initial vertex v_0 and assume $v_0 \in W_0$ (as we have argued, one can efficiently compute the winning regions). If desired, one can then repeat our procedure to compute winning strategies for all other vertices in W_0.

7.1.2 Implementation of Strategies and Strategy Automata

In order to solve a labeled safety game, Lemma 7.1 suggests to translate it into a classical safety game and apply the attractor computation of Section 2.3, which yields a positional winning strategy. One can then translate this strategy into a winning trace strategy. In order to do so, however, the player needs a mechanism to track the vertex a trace has currently reached because this information is no longer externally given (but needed in order to play the positional strategy).

A natural way to implement a positional strategy for Player σ is to use the arena as memory structure and delete all edges of Player σ vertices that are not picked by the strategy. The resulting trace strategy works as follows: if it is Player σ's turn after playing a finite trace $u \in \Sigma^*$, the player follows the u-labeled path in the arena (starting in the initial vertex) to determine the vertex reached by u and plays the action that labels the unique outgoing edge. In fact, it is already enough to keep the part of this restricted arena that can still be reached from the initial vertex. We denote the part of

an arena $\mathfrak{A}$ that is reachable from an initial vertex via a positional strategy f by $\mathfrak{A}_f$ and call it the *implementation of* f.

However, we pursue a different approach and aim at a broader class of trace strategies. Our idea is to represent a trace strategy in terms of a DFA that reads finite traces and accepts if and only if the input is a finite trace that is played according to the represented trace strategy. We call this kind of DFAs *strategy automata* and use the definition below. We demonstrate in Section 7.3 that strategy automata are often very compact implementations of trace strategies.

Definition 7.5 (Strategy automaton)**.** We call a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F_\mathcal{A})$ a *strategy automaton* for Player σ in the labeled safety game $\mathfrak{G} = (\mathfrak{A}, F)$ from vertex $v_0 \in V$ if its language fulfills the following conditions:

1. The language $L(\mathcal{A})$ is prefix-closed; that is, $\mathrm{Pref}(L(\mathcal{A})) \subseteq L(\mathcal{A})$.

2. For all $u \in L(\mathcal{A})$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$ and $v \in V_\sigma$, there exists an action $a \in \Sigma$ such that $(v, a, v') \in E$ and $ua \in L(\mathcal{A})$.

3. For all $u \in L(\mathcal{A})$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$ and $v \in V_\sigma$, the existence of an action $a \in \Sigma$ with $ua \in L(\mathcal{A})$ implies $(v, a, v') \in E$ for a suitable $v' \in V$.

4. For all $u \in L(\mathcal{A})$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$ and $v \in V_{1-\sigma}$, the condition $ua \in L(\mathcal{A})$ is satisfied for all $(v, a, v') \in E$.

We call a strategy automaton *winning for Player 0*—or a *winning strategy automaton*—if its language satisfies the following additional condition:

5. For all $u \in L(\mathcal{A})$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$, the condition $v \in F$ is satisfied.

Note that a strategy automaton is free to accept or reject words that do not form a finite trace in the arena.

Given a strategy automaton $\mathcal{A}$ for Player σ from a vertex $v_0 \in V$, we obtain a trace strategy $f_\mathcal{A}$ as follows. For a finite trace $u \in \Sigma^*$ with $\mathfrak{A}\colon v_0 \xrightarrow{u} v$ and $v \in V_\sigma$, we define:

- If $u \in L(\mathcal{A})$, then $f_\mathcal{A}(u) = a$ for an arbitrary but fixed $a \in \Sigma$ such that $ua \in L(\mathcal{A})$; due to Condition 2 of Definition 7.5, such a symbol always exists but may not be unique.

- If $u \notin L(\mathcal{A})$, then $f_\mathcal{A}(u) = a$ for an arbitrary but fixed $a \in \Sigma$ such that an edge $(v, a, v') \in E$ exists.

We also say that $\mathcal{A}$ *realizes* the strategy $f_\mathcal{A}$. In fact, a strategy automaton does not necessarily realize a unique strategy because it may allow more than one, and potentially not always the same, choice in a vertex of Player σ.

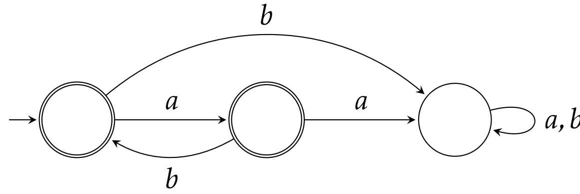


Figure 7.2: A strategy automaton that is winning for Player 0 in the labeled safety game of Figure 7.1 from vertex v_0.

Example 7.3. Let us again consider our running example of Figure 7.1 (on Page 211). The DFA depicted in Figure 7.2 is a strategy automaton that is winning for Player 0 from vertex v_0. A trace strategy derived from $\mathcal{A}$ is

$$f_{\mathcal{A}}(u) = \begin{cases} a & \text{if } |u| \text{ is even;} \\ b & \text{if } |u| \text{ is odd;} \end{cases}$$

where $u \in \{a, b\}^*$. ◁

Although a strategy automaton potentially realizes various different strategies, all of them are winning if the strategy automaton is winning.

Lemma 7.2. *Let $\mathfrak{G}$ be a labeled safety game, $\mathcal{A}$ a strategy automaton that is winning for Player 0 in $\mathfrak{G}$ from a vertex $v \in V$, and $f_{\mathcal{A}}$ a trace strategy realized by $\mathcal{A}$. Then, $f_{\mathcal{A}}$ is a winning trace strategy in $\mathfrak{G}$ for Player 0 from v.*

Proof of Lemma 7.2. A standard induction over the length of finite traces using Conditions 2 to 4 of Definition 7.5 shows that if a finite trace u is played according to $f_{\mathcal{A}}$, then $u \in L(\mathcal{A})$. This means in particular that whenever u reaches a vertex $v \in V_\sigma$, Player 0 can play an action $a \in \Sigma$ such that $ua \in L(\mathcal{A})$ holds. Since $\varepsilon \in L(\mathcal{A})$ (see Condition 1) and all finite traces $u \in L(\mathcal{A})$ stay inside F (see Condition 5), Player 0 can force to stay in F and, thus, wins from v. □

Clearly, not every trace strategy can be realized by a strategy automaton. However, the next definition shows that Definition 7.5 is sound in that for every labeled safety game (with finitely many vertices) and initial vertex in the winning region of Player 0 one can construct a strategy automaton that realizes a winning strategy from the initial vertex.

Definition 7.6 (Canonical strategy automaton)**.** Let $\mathfrak{G} = (\mathfrak{A}, F)$ be a labeled safety game over the Σ-arena $\mathfrak{A} = (V_0, V_1, E)$ and $v_0 \in W_0$ a vertex in the winning region of Player 0.

The *canonical strategy automaton* for $\mathfrak{G}$ and v_0 is the DFA $\mathcal{A}_{\mathfrak{G},v_0} = (Q, \Sigma, q_0, \delta, F')$ defined by

- $Q = W_0 \uplus \{q_s\}$ (q_s is a new state not contained in V);
- $q_0 = v_0$;
- $F' = W_0$; and
- $\delta(p,a) = \begin{cases} q & \text{if } p,q \in W_0 \text{ and } (p,a,q) \in E; \\ q_s & \text{otherwise;} \end{cases}$

where $p, q \in Q$ and $a \in \Sigma$.

It is not hard to verify that $\mathcal{A}_{\mathfrak{G},v_0}$ is a winning strategy automaton since it accepts the set of all finite prefixes of winning traces. Thus, one can view the canonical strategy automaton as a normal form for representing winning strategies in labeled safety games.

In fact, strategy automata are expressive enough to realize every positional strategy and every finite memory strategy[1], which subsume positional strategies. One can see this as follows: for a fixed initial vertex $v_0 \in V_\sigma$, a strategy f_σ induces a set of plays that is generated by the different choices of Player $1-\sigma$. If f_σ is a finite memory strategy, then the set of finite play prefixes is regular. Thus, the corresponding set of finite traces is also regular. One can construct a strategy automaton accepting this set by using a product of the arena and the Mealy machine underlying the finite memory strategy.

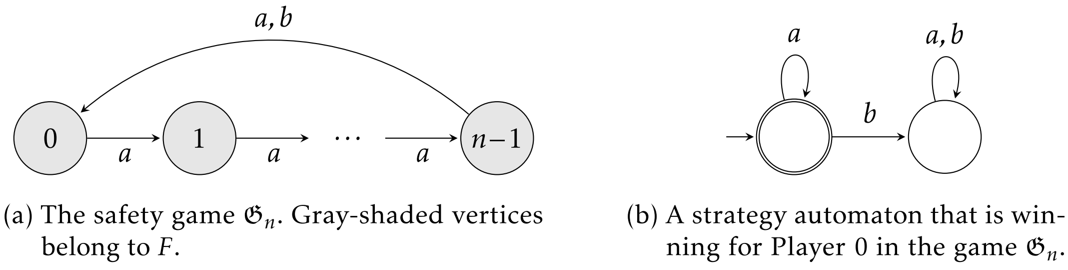
7.1.3 Size of Strategy Implementations

In order to compare different implementations of strategies, we define the *size* of an implementation. Since all established methods in the literature for computing strategies in safety games compute positional ones, we confine ourselves to comparing implementations of positional strategies and strategy automata.

Definition 7.7 (Size of a strategy implementation)**.**

- The size of a strategy automaton $\mathcal{A}$ is the number of $\mathcal{A}$'s states, denoted by $|\mathcal{A}|$.
- The size of the implementation of a positional strategy f is the number of vertices of $\mathfrak{A}_f$ (see Section 7.1.2), denoted by $|\mathfrak{A}_f|$.

[1] One usually defines finite memory strategies in terms of Mealy machines that read finite play prefixes and output a player's next move; see Grädel, Thomas, and Wilke [GTW02] for a thorough discussion.



(a) The safety game $\mathfrak{G}_n$. Gray-shaded vertices belong to F.

(b) A strategy automaton that is winning for Player 0 in the game $\mathfrak{G}_n$.

Figure 7.3: A family $(\mathfrak{G}_n)_{n\in\mathbb{N}}$ of safety games and a strategy automaton (of constant size) that realizes a winning strategy in the game $\mathfrak{G}_n$ for every $n \in \mathbb{N}$.

Strategy automata can in fact be succinct implementations of trace strategies. Let us formalize and prove this claim.

Lemma 7.3. *There exists a family $(\mathfrak{G}_n)_{n\in\mathbb{N}}$ of labeled safety games such that for every $n \in \mathbb{N}$*

- *the size of the implementation of any positional winning strategy for Player 0 in $\mathfrak{G}_n$ is n; whereas*
- *there exists a strategy automaton of size 2 that is winning for Player 0 in $\mathfrak{G}_n$ from every vertex $v \in V$.*

Proof of Lemma 7.3. Let $n \in \mathbb{N}$. We define $\mathfrak{G}_n = (\mathfrak{A}_n, F_n)$ to be the labeled safety game over the $\{a, b\}$-arena $\mathfrak{A}_n = (V_{0,n}, V_{1,n}, E_n)$ with

- $V_{0,n} = [n]$;
- $V_{1,n} = \emptyset$;
- $E_n = \{(i, a, (i+1) \bmod n) \mid i \in [n]\} \cup \{(n-1, b, 0)\}$; and
- $F_n = V_{0,n}$.

Figure 7.3a depicts the game $\mathfrak{G}_n$. Since all vertices are safe, $W_0 = V_{0,n}$ and $W_1 = \emptyset$ holds.

There exist exactly two positional (winning) strategies, let us call them f_0 and f_0', which differ only in the action that is played in vertex v_{n-1}; in any other vertex, Player 0 has to play the action a, and the play proceeds to the successor vertex. Hence, the implementations of both f and f_0' contain the whole arena, which implies $|\mathfrak{A}_{f_0}| = |\mathfrak{A}_{f_0'}| = |V_{0,n}| = n$.

On the other hand, the DFA depicted in Figure 7.3b is a strategy automaton that is winning for Player 0 in $\mathfrak{G}_n$ from every vertex $v \in V_{0,n}$. In contrast to $\mathfrak{A}_{f_0}$ and $\mathfrak{A}_{f_0'}$, this strategy automaton always has two states, independent of n. □

The implementation of a positional strategy in the proof of Lemma 7.3 suffers from the fact that it exactly remembers which vertex a finite trace has reached. This is clearly superfluous in the given example and exploited by the strategy automaton (in Section 7.3, we describe an approach to minimize the implementation of a positional strategy, which partly resolves this issue). Note, however, that the implementation of a strategy that plays action b in vertex $n-1$ cannot be reduced because action b has to be played at the exact right move. In fact, both computing a positional winning strategy that yields a minimal implementation and computing a minimal strategy automaton is hard. This is a consequence of a result by Ehlers [Ehl11], which we discuss in the next section in more detail.

7.1.4 Minimal Strategy Automata

In Section 7.2.1, we show how to check in polynomial time whether a given DFA is a strategy automaton (i.e., whether it satisfies Definition 7.5). However, constructing a minimal strategy automaton is a computationally hard task, as the next theorem states.

Theorem 7.4. *The decision problem* $\mathcal{P}_{min}$

> *"Given a labeled safety game* $\mathfrak{G}$*, a vertex* $v_0 \in V$*, and* $k \in \mathbb{N}$*. Does a strategy automaton with at most k states that realizes a winning strategy for Player 0 in* $\mathfrak{G}$ *from* v_0 *exist?"*

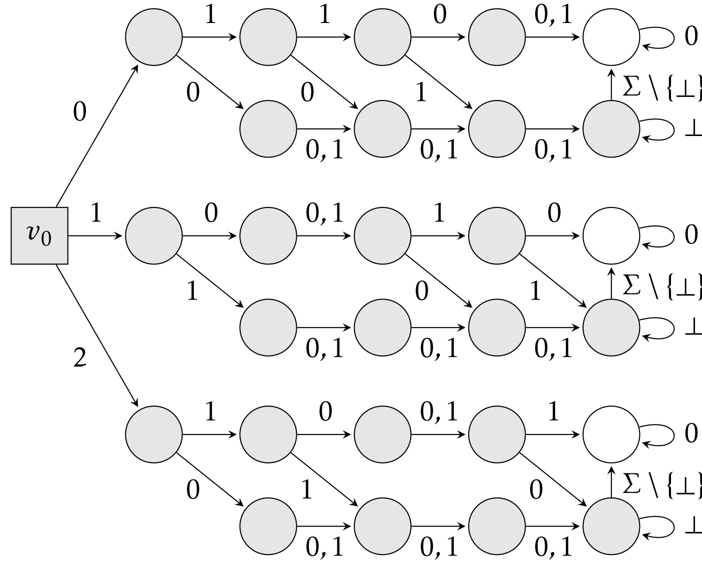
is NP*-complete.*

Theorem 7.4 is a consequence of a result by Ehlers [Ehl11]. In fact, Ehlers' result also implies that Theorem 7.4 already holds for labeled safety games that are played by only one player over a Σ-arena with $|\Sigma| = 2$. Moreover, a minimal strategy automaton is not even approximable within any polynomial. It is also worth noting that Kupferman et al. [KLVY11] studied a similar problem in the context of bounded synthesis, but their results cannot be transferred easily to the present setting.

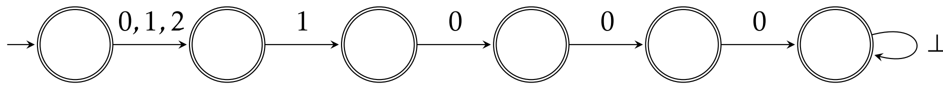
Since we are not interested in approximations, we here present a proof of Theorem 7.4 that is much simpler than Ehlers'. The key idea, which we borrow from Ehlers [Ehl11], is to reduce the problem of finding a model of a Boolean formula in 3-CNF[2] to the problem of finding a minimal strategy automaton. More precisely, from a formula φ in 3-CNF, we construct a safety game $\mathfrak{G}_\varphi$ over a Σ-arena of polynomial size in φ such that there exists a "small" strategy automaton that realizes a winning strategy for Player 0 in $\mathfrak{G}_\varphi$ from the initial vertex v_0 if and only if φ is satisfiable.

Before we give a formal definition of the safety game $\mathfrak{G}_\varphi$, let us illustrate the idea of its construction with an example.

[2] A formula is in 3-CNF if it is in CNF and every clause contains exactly three literals.



(a) The safety game $\mathfrak{G}_\varphi$ of Example 7.4. Vertices shaded gray belong to F.



(b) A strategy automaton that is winning for Player 0 in the safety game of Figure 7.4a from vertex v_0. Missing transitions point to a nonaccepting sink-state, which is not shown for the sake of readability.

Figure 7.4: The safety game $\mathfrak{G}_\varphi$ of Example 7.4 and a winning strategy automaton.

Example 7.4. Consider the 3-CNF formula

$$\varphi := (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor x_2 \lor \neg x_4),$$

which consists of $m = 3$ clauses and ranges over $n = 4$ variables.

The corresponding labeled safety game $\mathfrak{G}_\varphi$ is shown in Figure 7.4a. Its construction is as follows: The arena consists of one subgraph for each clause. In each subgraph, Player 0 can win by moving along a $\{0, 1\}$-labeled path (followed by $\perp^\omega$) that avoids the white vertex. Such a path corresponds to an interpretation of the variables $x_1, \ldots, x_n$ that satisfies the clause: the first move assigns a value to x_1, the second to x_2, and so on.

If φ is satisfiable, then there exists an interpretation of the variables that satisfies all clauses. From this interpretation, we can derive a strategy automaton with at most $n + 3$ states that plays according to the interpretation and, thus, avoids the white vertices, no matter what action Player 1 chooses in v_0 (cf. Figure 7.4b).

Contrary, if φ is unsatisfiable, then there are two subgraphs in which Player 0 needs to follow distinct paths to avoid the white vertices. However, a strategy automaton realizing such a trace strategy needs strictly more than $n+3$ states. To see why, we observe that each winning trace has a prefix of $n+1$ symbols different from $\bot$, which is followed by $\bot^\omega$. To verify this property, a strategy automaton needs at least $n+3$ states: $n+2$ states to process valid traces and an additional sink state to handle invalid traces. Moreover, the strategy has to play differently in at least two distinct subgraphs, which in turn requires at least one additional state to distinguish these cases. ◁

Let us now define the safety game $\mathfrak{G}_\varphi$ formally. To this end, let

$$\varphi(x_1, \dots, x_n) \coloneqq \bigwedge_{i=0}^{m-1} C_i$$

be a 3-CNF formula with $m \geq 2$ clauses over $n \geq 3$ variables where each clause

$$C_i \coloneqq (l_{i,1} \vee l_{i,2} \vee l_{i,3})$$

consists of three literals $l_{i,j} \in \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}$; we assume without loss of generality that no variable occurs twice in a clause. Letting $\Sigma = [m] \cup \{\bot\}$, we define the Σ-arena $\mathfrak{A}_\varphi = (V_0, V_1, E)$ and the safety game $\mathfrak{G}_\varphi = (\mathfrak{A}_\varphi, F)$ by

- $V_0 = \{(i,j,k) \mid 0 \leq i < m, 1 \leq j \leq n+1, k \in \{0,1\}\} \setminus \{(i,1,1) \mid 0 \leq i < m\}$;
- $V_1 = \{v_0\}$;
- $F = V \setminus \{(i, n+1, 0) \mid 0 \leq i < m\}$; and
- $E \subseteq V \times \Sigma \times V$ is the smallest relation that satisfies the following:[3]
  - $\big(v_0, i, (i,1,0)\big) \in E$ for all $0 \leq i < m$;
  - if $x_j \in C_i$, then $\big((i,j,0), 0, (i,j+1,0)\big) \in E$ as well as $\big((i,j,0), 1, (i,j+1,1)\big) \in E$ for all $0 \leq i < m$ and $1 \leq j \leq n$;
  - if $\neg x_j \in C_i$, then $\big((i,j,0), 0, (i,j+1,1)\big) \in E$ as well as $\big((i,j,0), 1, (i,j+1,0)\big) \in E$ for all $0 \leq i < m$ and $1 \leq j \leq n$;
  - if both $x_j \notin C_i$ and $\neg x_j \notin C_i$, then $\big((i,j,0), 0, (i,j+1,0)\big) \in E$ as well as $\big((i,j,0), 1, (i,j+1,0)\big) \in E$ for all $0 \leq i < m$ and $1 \leq j \leq n$;
  - $\big((i,j,1), 0, (i,j+1,1)\big) \in E$ as well as $\big((i,j,1), 1, (i,j+1,1)\big) \in E$ for all $0 \leq i < m$ and $2 \leq j \leq n$;

[3] We write $l_i \in C_j$ to denote that the literal l_i occurs in the clause C_j.

- $\big((i,n+1,0),0,(i,n+1,0)\big) \in E$ for all $0 \leq i < m$;
- $\big((i,n+1,1),\bot,(i,n+1,1)\big) \in E$ as well as $\big((i,n+1,1),a,(i,n+1,0)\big) \in E$ for all $0 \leq i < m$ and $a \in \Sigma \setminus \{\bot\}$.

Note that the arena $\mathfrak{A}_\varphi$ (and, hence, the game $\mathfrak{G}_\varphi$) is of size polynomial in m and n. Moreover, the next lemma states that $\mathfrak{G}_\varphi$ has indeed the desired properties.

Lemma 7.5. *Let φ be a 3-CNF formula with at least two clauses ranging over $n \geq 3$ variables. If φ is satisfiable, then there exists a strategy automaton with $n+3$ states that realizes a winning trace strategy for Player 0 in $\mathfrak{G}_\varphi$ from vertex v_0. If φ is unsatisfiable, then any such winning strategy automaton has strictly more than $n+3$ states.*

Proof of Lemma 7.5. Let φ be as in Lemma 7.5. We first observe that Player 0 can win from vertex v_0 independently of whether φ is satisfiable or not.

To prove the properties claimed in Lemma 7.5, we first assume that φ is satisfiable with model $\mathfrak{M}$. Based on $\mathfrak{M}$, we construct the strategy automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, F_\mathcal{A})$ with $Q = \{q_0, \ldots, q_{n+1}, q_s\}$, $F_\mathcal{A} = Q \setminus \{q_s\}$, and

$$\delta(p,a) = \begin{cases} q_1 & \text{if } p = q_0 \text{ and } a \in [m]; \\ q_{i+1} & \text{if } p = q_i,\ i \in \{1,\ldots,n\},\ x_i^{\mathfrak{M}} = \mathit{true}, \text{ and } a = 1; \\ q_{i+1} & \text{if } p = q_i,\ i \in \{1,\ldots,n\},\ x_i^{\mathfrak{M}} = \mathit{false}, \text{ and } a = 0; \\ q_{n+1} & \text{if } p = q_{n+1} \text{ and } a = \bot; \\ q_s & \text{otherwise;} \end{cases}$$

where $p \in Q$ and $a \in \Sigma$. First, we observe that $\mathcal{A}$ has $n+3$ states. Second, every trace strategy $f_\mathcal{A}$ plays according to the interpretation of the variables given by $\mathfrak{M}$. Since $\mathfrak{M}$ is a model for every clause of φ and by construction of $\mathfrak{G}_\varphi$, every trace played according to any trace strategy $f_\mathcal{A}$ is winning, no matter what action Player 1 plays in v_0. Thus, $\mathcal{A}$ is a winning strategy automaton with $n+3$ states.

Let φ now be unsatisfiable. Towards a contradiction, assume that there exists a strategy automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, F_\mathcal{A})$ with (at most) $n+3$ states that is winning for Player 0 in $\mathfrak{G}_\varphi$ from v_0. Moreover, fix an arbitrary trace strategy $f_\mathcal{A}$ realized by $\mathcal{A}$. By Lemma 7.2 (on Page 215), every trace strategy realized by $\mathcal{A}$, and $f_\mathcal{A}$ in particular, is winning for Player 0. However, we show next that a winning strategy automaton needs strictly more than $n+3$ states in order for $f_\mathcal{A}$ to be winning.

We first observe that all winning traces for Player 0 starting in v_0 are of the form $ka_1 \ldots a_n \bot^\omega$ with $k \in [m]$ and $a_i \in \{0,1\}$ for $i \in \{1,\ldots,n\}$. In other words, Player 1 chooses action k in v_0 and Player 0 subsequently plays the actions a_1 to a_n, followed by playing action $\bot$ ad infinitum. By construction of $\mathfrak{G}_\varphi$, the actions $a_1,\ldots,a_n$ correspond to an interpretation of the variables $x_1,\ldots,x_n$ that satisfies the clause C_k.

Now, let $\tau_{v_0} = 0a_1 \dots a_n \bot^\omega$ be a trace starting in v_0 that is played according to $f_\mathcal{A}$. Since $f_\mathcal{A}$ is a winning trace strategy, the actions $a_1, \dots, a_n$ correspond to an interpretation that satisfies the clause C_0. However, φ is unsatisfiable. This implies that there exists a $j \in \{1, \dots n\}$ such that the interpretation corresponding to $a_1, \dots, a_n$ does not satisfy clause C_j and the trace $ja_0 \dots a_n \bot^\omega$ is not winning for Player 0. On the other hand, since $f_\mathcal{A}$ is winning for Player 0, there exists a winning trace $\tau'_{v_0} = ja'_1 \dots a'_n \bot^\omega$ that is played according to $f_\mathcal{A}$. This trace necessarily satisfies $a_i \neq a'_i$ for an $i \in \{1, \dots, n\}$.

Next, consider the runs

$$\mathcal{A}\colon q_0 \xrightarrow{0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_{n+1} \xrightarrow{\bot} q_{n+2}$$

and

$$\mathcal{A}\colon q_0 \xrightarrow{j} q'_1 \xrightarrow{a'_1} q'_2 \xrightarrow{a'_2} \dots \xrightarrow{a'_n} q'_{n+1} \xrightarrow{\bot} q'_{n+2}.$$

For one thing, we observe that the states $q_0, \dots, q_{n+1}$ are pairwise distinct. To see why, assume the converse and let $i, j \in [n+2]$, $i < j$, such that $q_i = q_j$; in this case, $\mathcal{A}$ also realizes a trace strategy that continues the trace $0a_0 \dots a_{i-1}$ by playing $a_j \dots a_n \bot^\omega$, which is not winning for Player 0 (as the action $\bot$ is played too early). Moreover, $\{q_0, \dots, q_{n+1}\} \subseteq F_\mathcal{A}$ since $0a_0 \dots a_n$ is played according to $f_\mathcal{A}$.

For another, we deduce $q_1 \neq q'_1$; if this does not hold, $\mathcal{A}$ realizes a trace strategy that plays the trace $ja_0 \dots a_n \bot^\omega$, which we know is not winning for Player 0. The state q'_1 is also distinct from any of the states $q_0, q_2, \dots, q_n$, which one can establish using the same argument that shows that $q_0, \dots, q_{n+1}$ are pairwise distinct. Moreover, $q'_1 \in F_\mathcal{A}$ because $ja'_0 \dots a'_n$ is played according to $f_\mathcal{A}$.

Finally, $\mathcal{A}$ needs at least one nonaccepting state, say $q \notin F$, since it is a winning strategy automaton. If such a state does not exist, $\mathcal{A}$ realizes every possible trace strategy, some of which are not winning for Player 0 (as they correspond to unsatisfying interpretations). This is clearly a contradiction.

In summary, we proved that $\mathcal{A}$ has to consist of strictly more than $n+3$ states, namely at least $q_0, \dots, q_{n+1}, q'_1$, and q. This yields the desired contradiction. □

We can now prove Theorem 7.4.

Proof of Theorem 7.4. Lemma 7.5 proves that the decision problem $\mathcal{P}_{\min}$ is NP-hard. A nondeterministic algorithm that guesses a DFA of size at most k and verifies in polynomial time that the guessed DFA is a strategy automaton proves that $\mathcal{P}_{\min}$ is also in NP (we present a polynomial time algorithm for the latter task in Section 7.2.1). Therefore, $\mathcal{P}_{\min}$ is NP-complete. □

7.2 Learning Small Strategy Automata

After introducing the setting, we can now turn to learning strategy automata. Our algorithm exploits the fact that active learning algorithms typically conjecture DFAs that monotonically grow in size during the learning process. Given a labeled safety game $\mathfrak{G}$ and a vertex $v_0 \in W_0$, the key idea is to let the teacher teach the language $L(\mathcal{A}_{\mathfrak{G},v_0})$ of the canonical strategy automaton $\mathcal{A}_{\mathfrak{G},v_0}$ and stop the learning prematurely once the learner has conjectured a strategy automaton that is winning for Player 0; that is, the teacher replaces a classical equivalence query with the check whether the given conjecture is a winning strategy automaton. In this way, the teacher "guides" the learner to come up with a small winning strategy automaton (which immediately passes an equivalence query) before the learner eventually learns the automaton $\mathcal{A}_{\mathfrak{G},v_0}$ as last resort. Since the teacher stops the learning as soon as possible, the resulting DFA is guaranteed to be a winning strategy automaton that is not larger than $\mathcal{A}_{\mathfrak{G},v_0}$.

In order to make this procedure work, a teacher has to satisfy the following definition.

Definition 7.8 (Teacher for labeled safety games)**.** Given a labeled safety game $\mathfrak{G} = (\mathfrak{A}, F)$ over a Σ-arena $\mathfrak{A}$ and a vertex $v_0 \in V$, a teacher for $\mathfrak{G}$ answers queries as follows:

Membership query On a membership query with a word $u \in \Sigma^*$, the teacher returns "yes" if $u \in L(\mathcal{A}_{\mathfrak{G},v_0})$ and "no" otherwise.

Equivalence query On an equivalence query with a DFA $\mathcal{A}$, the teacher checks whether $\mathcal{A}$ is a winning strategy automaton for Player 0 in $\mathfrak{G}$ from v_0 (i.e., whether $\mathcal{A}$ satisfies Definition 7.5; see Page 214). If this is the case, he returns "yes". If this is not the case, he returns a counterexample proving that $\mathcal{A}$ violates Definition 7.5 (we describe in Section 7.2.1 how to do this).

Algorithm 7.1 (on Page 224) lists our heuristic in pseudo code. In order to learn $L(\mathcal{A}_{\mathfrak{G},v_0})$, we first compute the winning region W_0 (e.g., using the standard attractor computation); note that we are here not interested in learning a winning strategy automaton in the most efficient way but in learning a small one. If $v_0 \in W_0$, we set up a teacher according to Definition 7.8 and plug in an active learning algorithm that produces conjectures that grow monotonically in size. Once the learning has finished, we return the resulting DFA, which is then guaranteed to be a winning strategy automaton for Player 0 in $\mathfrak{G}$ from v_0.

However, the quality (i.e., the size) of the results of Algorithm 7.1 mainly depends on two aspects: the "quality" of counterexamples the teacher returns and the particular choice of the learning algorithm.

Algorithm 7.1: Learning-based algorithm for synthesizing strategy automata.

Input: A labeled safety game $\mathfrak{G} = (\mathfrak{A}, F)$ over a Σ-arena $\mathfrak{A}$ and an initial vertex $v_0 \in V$.

1 Compute the winning region W_0.
2 **if** $v_0 \notin W_0$ **then**
3 | **return** that no winning strategy automaton for Player 0 from v_0 exists.
4 **end**
5 Construct the canonical strategy automaton $\mathcal{A}_{\mathfrak{G},v_0}$ and a teacher according to Definition 7.8.
6 Run an active learning algorithm that conjectures DFAs of increasing size.
7 **return** the learned DFA.

The question how to compute "good" counterexamples is beyond the scope of this thesis because this largely depends on domain-specific characteristics of the game at hand. However, in Section 7.2.1, we present a generic teacher, which works for every labeled safety game and computes counterexamples that are minimal with respect to the canonical order on words. The hope is that short counterexamples prevent the learner from constructing unnecessary large conjectures.

Regarding the choice of appropriate learning algorithms, it turned out that the standard active learning algorithms described in Section 3.2 tend to learn $\mathcal{A}_{\mathfrak{G},v_0}$ exactly. Therefore, we develop modifications of both Angluin's as well as Kearns and Vazirani's learning algorithms, which avoid this issue and performed better in our experiments. Section 7.2.2 presents these modifications in detail.

In Section 7.2.3, we conclude by proving the correctness of Algorithm 7.1 and commenting on its time and space complexity.

7.2.1 A Teacher for Strategy Automata

We now describe how to implement a teacher for safety games according to Definition 7.8 and briefly comment on the time and space needed to answer queries. We assume that the teacher has access to the canonical strategy automaton $\mathcal{A}_{\mathfrak{G},v_0}$.

Answering membership queries The teacher answers membership queries with respect to $L(\mathcal{A}_{\mathfrak{G},v_0})$; that is, on a membership query with a word $u \in \Sigma^*$, the teacher simulates the run of $\mathcal{A}_{\mathfrak{G},v_0}$ on u and returns "yes" if $u \in L(\mathcal{A}_{\mathfrak{G},v_0})$ and "no" otherwise.

Answering equivalence queries In the present setting, the equivalence query is replaced with a "correctness test"; that is, on an equivalence queries with the DFA

$\mathcal{A} = (Q, \Sigma, q_0, \delta, F_\mathcal{A})$, the teacher does not check whether $L(\mathcal{A}) = L(\mathcal{A}_{\mathfrak{G},v_0})$ holds but whether $\mathcal{A}$ satisfies Definition 7.5.

In order to do so, the teacher first checks whether $\varepsilon \in L(\mathcal{A})$ holds (see Condition 1 of Definition 7.5). If this is not the case, he returns ε as counterexample.

Next, the teacher checks whether $L(\mathcal{A})$ is prefix closed (also, see Condition 1 of Definition 7.5). $L(\mathcal{A})$ is not prefix-closed if there are two words u, ua such that $u \notin L(\mathcal{A})$ and $ua \in L(\mathcal{A})$. The teacher can easily identify such words using a breadth-first search in the transition structure of $\mathcal{A}$ starting in q_0. Since $L(\mathcal{A}_{\mathfrak{G},v_0})$ is prefix-closed, either u or ua is classified incorrectly by $\mathcal{A}$. The teacher checks which one by simulating the runs of $\mathcal{A}_{\mathfrak{G},v_0}$ on both u and ua and returns the respective word as counterexample.

To check the remaining conditions (i.e., Conditions 2 to 5 of Definition 7.5), the teacher constructs what we call the *product of the arena* $\mathfrak{A}$ *and the conjecture* $\mathcal{A}$. This product is the DFA $\mathfrak{A} \times \mathcal{A} = (Q', \Sigma, q_0', \delta', F')$ where

- $Q' = (V \cup \{v_s\}) \times Q$ (v_s is a new sink-vertex not contained in V);
- $q_0' = (v_0, q_0)$;
- F' is unimportant and can be arbitrary;
- $\delta'((v,q),a) = \begin{cases} (v', \delta(q,a)) & \text{if a (unique) } v' \in V \text{ with } (v,a,v') \in E \text{ exists;} \\ (v_s, q) & \text{otherwise;} \end{cases}$

  and $\delta'((v_s,q),a) = (v_s,q)$ where $v \in V$, $q \in Q$, and $a \in \Sigma$.

Note that whenever there is no edge (v,a,v') in the arena, the transition $\delta'((v,q),a)$ in the product points to the sink-state (v_s,q).

Starting in q_0', the teacher now performs a breadth-first search. For each state of the product reached during the search, one can locally check whether one of Conditions 2 to 5 of Definition 7.5 is violated. If the teacher detects a violation, he derives a counterexample from a word that leads to the violation and returns it. If the conjecture is a strategy automaton that is winning for Player 0, the search terminates after visiting all states in the product. In this case, the teacher returns "yes".

Note that a breadth-first search that explores transitions with respect to a total order of the actions finds the canonically smallest counterexample (if one exists).

Runtime of the teacher To construct the teacher described above, we first need to construct the automaton $\mathcal{A}_{\mathfrak{G},v_0}$. An efficient implementation can compute W_0 in time linear in $|E| \in \mathcal{O}(|V| \cdot |\Sigma|)$ and, hence, the construction of $\mathcal{A}_{\mathfrak{G},v_0}$ falls in the same complexity class. The automaton $\mathcal{A}_{\mathfrak{G},v_0}$ has size $|W_0| + 1 \in \mathcal{O}(V)$.

Once the teacher has been constructed, he answers a membership query with a word $u \in \Sigma^*$ in time $\mathcal{O}(|u|)$.

The time needed to answer an equivalence query with a conjecture $\mathcal{A}$ is dominated by the construction of $\mathfrak{A} \times \mathcal{A}$ and the subsequent breadth-first search. The product is of size $|V| \cdot |\mathcal{A}|$, and a depth-first search is linear in the size of the product. If necessary, the teacher computes a counterexample on-the-fly, whose length can, thus, be bounded by $|\mathfrak{A} \times \mathcal{A}|$. Hence, the teacher answers equivalence queries in time $\mathcal{O}(|V| \cdot |\mathcal{A}|)$.

7.2.2 Two Domain-specific Learning Algorithms

The original active learning algorithms of Section 3.2 have in common that, once a new state of the automaton to learn has been discovered, all outgoing transitions of this state are examined. In our setting, however, this behavior is often undesirable. Consider, for instance, a situation in which a finite prefix $u \in \Sigma^*$ of a winning trace reaches the vertex $v \in V_0$ and the run of the current conjecture on u ends in state q. In such a situation, one outgoing transition from state q would suffice to successfully continue playing. However, as soon as the original learning algorithms identify a new state in the course of the learning process, they examine all outgoing transitions. In other words, these algorithms not only consider one way to continue playing but all. This behavior often results in learning the language $L(\mathcal{A}_{\mathfrak{G},v_0})$ exactly.

In order to avoid this undesirable behavior, we develop adaptations of Angluin's as well as Kearns and Vazirani's algorithms. The underlying idea is to probe for a transition only if the teacher provides evidence that a transition is indeed necessary. If such information is not available, we direct a transition to a nonaccepting sink-state. Simply put, we start with the assumption that all transitions lead to a nonaccepting sink-state and change our mind only if the teacher provides appropriate evidence. Since Algorithm 7.1 terminates the learning as soon as possible, the hope is that such modified learning algorithms learn small winning strategy automata with only few transitions not leading to the sink-state.

At this point, it is important to emphasize that our adaptations are full-fledged active learning algorithms; that is, when used in the original active learning setting as introduced in Definition 3.7 (on Page 63), they learn the canonical minimal DFA for the target language and enjoy the same properties as the original active learning algorithms of Section 3.2 (except for a higher number of queries). Only when stopped prematurely during the learning process (e.g., as in the context of this chapter, where the equivalence query is replaced by a "correctness test"), unknown transitions lead to a dedicated nonaccepting sink-state.

For the remainder of this section, we consider the original active learning setting of Definition 3.7 rather than the special scenario of labeled safety games. In particular, let $L \subseteq \Sigma^*$ be a regular language—the target language—over a fixed alphabet Σ and $\mathcal{A}_L = (Q_L, \Sigma, q_0^L, \delta_L, F_L)$ the canonical minimal DFA accepting L.

A Variant of Angluin's Algorithm

Our variant of Angluin's algorithm works as Angluin's original algorithm described in Section 3.2.1 but drops the closedness constraint of observation tables and derives a conjecture as soon as the table becomes consistent. If a table O is consistent but not closed (i.e., information about one or more transitions is missing), we construct a conjecture, denoted by $\mathcal{A}'_O$, in which unknown transitions lead to a nonaccepting sink-state $\bot$. An equivalence query with $\mathcal{A}'_O$ then either happens to pass or provides new information. This information allows introducing a previously unknown transition or identifying a new L-equivalence class.

Given a consistent (but not necessarily closed) observation table $O = (R, S, T)$, we define the DFA $\mathcal{A}'_O = (Q_O, \Sigma, q_0^O, \delta_O, F_O)$ by

- $Q_O = \{[\![u]\!]_O \mid u \in R\} \uplus \{\bot\}$;
- $q_0^O = [\![\varepsilon]\!]_O$;
- $F_O = \{[\![u]\!]_O \mid u \in R, T(u) = 1\}$;
- $\delta_O([\![u]\!]_O, a) = \begin{cases} [\![va]\!]_O & \text{if there exists a } va \in R \text{ such that } u \sim_O v; \\ \bot & \text{otherwise;} \end{cases}$

  and $\delta(\bot, a) = \bot$ where $u \in R$ and $a \in \Sigma$.

It is not hard to verify that $\mathcal{A}'_O$, and δ in particular, is well-defined because O is consistent. Moreover, note that we define the transition $\delta([\![u]\!]_O, a)$ as soon as there exists an O-equivalent representative $v \in [\![u]\!]_O$ with $va \in R$ even if ua itself is not a representative.

It might happen that there exists a state $[\![u]\!]_O \notin F_O$ whose outgoing transitions loop back to $[\![u]\!]_O$, lead to $\bot$, or both. In this case, the state $\bot$ is superfluous, and we merge $[\![u]\!]_O$ and $\bot$ by removing the state $\bot$ and redirecting its incoming transitions to $[\![u]\!]_O$. Moreover, we also remove the state $\bot$ if it is not reachable from q_0^O. Note that state $\bot$ vanishes in both cases.

In analogy to Lemma 3.7 (on Page 65), we observe that the automaton $\mathcal{A}'_O$ classifies the representatives correctly.

Lemma 7.6. *Let $O = (R, S, T)$ be a consistent observation table for a regular language $L \subseteq \Sigma^*$ and $\mathcal{A}'_O$ as defined above. Then, $\mathcal{A}'_O$ is correct on all representatives $u \in R$ (i.e., $\mathcal{A}'_O$ satisfies $u \in L(\mathcal{A}'_O)$ if and only if $u \in L$ for all $u \in R$).*

One can prove Lemma 7.6 in the same way as Lemma 3.7. The idea is to exploit that R is prefix closed, which implies that only transitions of the form $\delta_O([\![v]\!]_O, a) = [\![va]\!]_O$ for $v, va \in R$ are used in a run of $\mathcal{A}'_O$ on a representative $u \in R$. In fact, the proof of

Lemma 3.7 does not rely the fact that the table is closed but only that R is prefix-closed and δ_O well-defined. Since this is also true in the present setting, the proof of Lemma 3.7 also proves Lemma 7.6.

In addition to Lemma 7.6, $\mathcal{A}'_O$ is isomorphic to $\mathcal{A}_L$ if O contains enough information in that at least one representative of every $\sim_L$-equivalence class is present in R (i.e, $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$ is satisfied).

Lemma 7.7. *Let $O = (R, S, T)$ be a consistent observation table for a regular language $L \subseteq \Sigma^*$. If $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$, then $\mathcal{A}'_O$ is isomorphic to $\mathcal{A}_L$.*

Proof of Lemma 7.7. This proof is similar to the proof of Lemma 3.8 (on Page 65). For the reader's convenience, however, we provide the full proof here. Let O be consistent, $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$, and $\mathcal{A}'_O$ be the DFA derived from O.

We first observe that the sink-state $\bot$ has been removed from the DFA $\mathcal{A}'_O$: since $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$, all traditions of $\mathcal{A}'_O$ are of the form $\delta_O(\llbracket u \rrbracket_O, a) = \llbracket ua \rrbracket_O$ for $u, ua \in R$. Thus, the state $\bot$ is unreachable from $q_O^0 = \llbracket \varepsilon \rrbracket_O$ and, therefore, removed in the preprocessing step.

Next, we show by induction over the length of words $u \in \Sigma^*$ that $\mathcal{A}'_O$ satisfies $\mathcal{A}'_O \colon \llbracket \varepsilon \rrbracket_O \xrightarrow{u} \llbracket u' \rrbracket_O$ with $u' \in R$ and $u' \sim_L u$. This fact implies $L(\mathcal{A}'_O) = L$ because

$$\begin{aligned} u \in L(\mathcal{A}'_O) &\Leftrightarrow \mathcal{A}'_O \colon \llbracket \varepsilon \rrbracket_O \xrightarrow{u} \llbracket u' \rrbracket_O \text{ with } u' \in R \text{ and } \llbracket u' \rrbracket_O \in F_O \\ &\Leftrightarrow T(u') = 1 \\ &\Leftrightarrow u' \in L \\ &\Leftrightarrow u \in L. \end{aligned}$$

Base case Let $u = \varepsilon$. Then, $\mathcal{A}'_O \colon \llbracket \varepsilon \rrbracket_O \xrightarrow{\varepsilon} \llbracket \varepsilon \rrbracket_O$ holds by definition of runs. Since $\sim_L$ is a congruence (and in particular reflexive), $\varepsilon \sim_L \varepsilon$ holds.

Induction step Let $u = va$, and consider the run of $\mathcal{A}'_O$ on v. Since $\bot$ has been removed from $\mathcal{A}'_O$, the run is of the form

$$\mathcal{A}'_O \colon \llbracket \varepsilon \rrbracket_O \xrightarrow{v} \llbracket v' \rrbracket_O \xrightarrow{a} \llbracket v'' \rrbracket_O$$

with $v', v'' \in R$. Applying the induction hypothesis now yields $v' \sim_L v$. Since $\sim_L$ is a congruence, we also know that $v'a \sim_L va$ holds. Furthermore, $v'a \sim_O v''$ holds by construction of $\mathcal{A}_O$ because the transition $\delta_O(\llbracket v' \rrbracket_O, a) = \llbracket v'' \rrbracket_O$ was used in the run. Since $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L)$ (and since $u \sim_L v$ implies $u \sim_O v$ for every $u, v \in R$), we obtain $v'a \sim_L v''$. In total, $u = va \sim_L v'a \sim_L v''$ holds.

In conclusion, the fact that $\mathcal{A}'_O$ is isomorphic to $\mathcal{A}_L$ follows from $L(\mathcal{A}'_O) = L$ and $|A'_O| = \text{index}(\sim_O \cap R \times R) = \text{index}(\sim_L) = |\mathcal{A}_L|$. □

Algorithm 7.2: A variant of Angluin's active learning algorithm.

Input: A teacher for a regular language $L \subseteq \Sigma^*$.

1 Initialize the observation table $O = (R, S, T)$ with $R = S = \{\varepsilon\}$ and *update*(O).
2 **repeat**
3   **while** *O is not consistent* **do**
4   Pick $u \sim_O v \in R$, $w \in S$, and $a \in \Sigma$ with $T(uaw) \neq T(vaw)$.
5         $S \leftarrow S \cup \{aw\}$.
6   *update*(O).
7   **end**
8   Construct $\mathcal{A}'_O$, and perform an equivalence query with $\mathcal{A}'_O$.
9   **if** *the teacher returns a counterexample u* **then**
10         $R \leftarrow R \cup \mathrm{Pref}(u)$.
11   *update*(O).
12   **end**
13 **until** *the teacher returns "yes" on an equivalence query with $\mathcal{A}'_O$*.
14 **return** $\mathcal{A}'_O$.

Algorithm 7.2 lists our variant of Angluin's algorithm in pseudo code, which works like Angluin's original algorithm except for the closedness check and a modified way to derive conjectures.

Before we state the key properties of Algorithm 7.2, let us remind the reader that we consider the original active learning setting but not the case that the learning is stopped prematurely.

Theorem 7.8. *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Algorithm 7.2 learns a DFA isomorphic to the canonical minimal DFA $\mathcal{A}_L$ in time polynomial in the size n of $\mathcal{A}_L$ and the length m of the longest counterexample returned by the teacher. It asks $\mathcal{O}(n)$ equivalence queries and $\mathcal{O}(mn^2)$ membership queries.*

To prove Theorem 7.8, we first introduce a few auxiliary definitions. Given a target language $L \subseteq \Sigma^*$ and an observation table $O = (R, S, T)$, which is filled by querying a teacher for L, we say that O *contains the transition $\delta_L([\![u]\!]_L, a)$ of $\mathcal{A}_L$* if there exists a representative $va \in R$ such that $u \sim_L v$. Moreover, let $t(O)$ denote the number of transitions that are contained in O. Intuitively, one can think of $t(O)$ as a measure that counts how many transitions of $\mathcal{A}_L$ have already been discovered. Since $\mathcal{A}_L$ contains $\mathrm{index}(\sim_L)$ states and $\mathrm{index}(\sim_L) \cdot |\Sigma|$ transitions, $t(O)$ is bounded by $\mathrm{index}(\sim_L) \cdot |\Sigma|$.

Proof of Theorem 7.8. The proof is similar to the correctness proof of Angluin's algorithm (see the proof of Theorem 3.9 on Pages 69 and following). We split the proof into two parts: we first show that Algorithm 7.2 terminates and returns a DFA isomorphic to $\mathcal{A}_L$; then, we estimate the number of queries asked during the learning.

Let $n = \text{index}(\sim_L)$.

Correctness First, recall that $\text{index}(\sim_O \cap R \times R) \leq \text{index}(\sim_O) \leq n$ is always true because $u \sim_L v$ implies $u \sim_O v$ for all $u, v \in \Sigma^*$. Next, we make the following observations.

1. If O is not consistent, then Algorithm 7.2 adds a new separating word aw to S such that two previously O-equivalent representatives are now separated by the word aw. Hence, $\text{index}(\sim_O \cap R \times R)$ increases.
2. After adding a counterexample u and all of its prefixes to R, one or more of the following cases occur. Let $O = (R, S, T)$ the observation table before and $O' = (R', S', T')$ the observation table after u has been processed.
   a) O' is not consistent.
   b) $t(O)$ increased (i.e., $t(O) < t(O')$).
   c) $\text{index}(\sim_O \cap R \times R)$ increased (i.e., $\text{index}(\sim_O \cap R \times R) < \text{index}(\sim_{O'} \cap R' \times R')$).

   To prove that these are the only cases that can occur, we show that Case 2c has to occur whenever Cases 2a and 2b do not occur. To this end, assume that both Case 2a and Case 2b do not occur. By Lemma 7.6, we know that $\mathcal{A}'_{O'}$ classifies the counterexample u correctly since it has been added to R. However, $\text{index}(\sim_O \cap R \times R) = \text{index}(\sim_{O'} \cap R' \times R')$ and $t(O) = t(O')$ imply that neither new information about states nor about transitions is available; in this case, Algorithm 7.2 constructs the same conjecture again. Thus, $L(\mathcal{A}'_O) = L(\mathcal{A}'_{O'})$, which is a contradiction because u is a counterexample witnessing a difference between $\mathcal{A}'_O$ and $\mathcal{A}'_{O'}$.

Observation 1 implies that O is consistent once $\text{index}(\sim_O \cap R \times R) = n$; otherwise, Algorithm 7.2 extends the table and $\text{index}(\sim_O \cap R \times R)$ increases, which contradicts the upper bound $\text{index}(\sim_O \cap R \times R) \leq \text{index}(\sim_O) \leq n$. As long as $|\mathcal{A}'_O| < n$, the DFA $\mathcal{A}'_O$ necessarily accepts a language different from L and the teacher returns a counterexample. Moreover, Observation 2 in combination with Observation 1 implies that either $\text{index}(\sim_O \cap R \times R)$ or $t(O)$ increases between consecutive equivalence queries; in particular, $\text{index}(\sim_O \cap R \times R)$ increases once $t(O) = \text{index}(\sim_O \cap R \times R) \cdot |\Sigma|$. In the worst case, Algorithm 7.2 proceeds until $\text{index}(\sim_O \cap R \times R) = n$. Once this has happened, the observation table is consistent, and Lemma 7.7 asserts that $\mathcal{A}'_O$ is isomorphic to $\mathcal{A}_L$. Then, $\mathcal{A}'_O$ passes an equivalence query and Algorithm 7.2 terminates and returns a DFA that is isomorphic to $\mathcal{A}_L$.

However, it might happen that a conjecture passes an equivalence query before $\text{index}(\sim_O \cap R \times R) = n$ holds. In this case, we first observe $|\mathcal{A}'_O| \geq n$ since the loop guard in Line 13 of Algorithm 7.2 asserts $L(\mathcal{A}'_O) = L$ and every DFA accepting L has at least

n states. Moreover, we argue that $|\mathcal{A}'_O| \leq n$ is also satisfied and, therefore, $|\mathcal{A}'_O| = n$. To do so, we distinguish two cases:

- The state $\bot$ has been merged or removed. In this case, the claim immediately holds because $|\mathcal{A}'_O| = \text{index}(\sim_O \cap R \times R) \leq n$.

- The state $\bot$ has not been merged and not been removed. In this case, there exists a transition $\delta_O([\![u]\!]_O, a) = \bot$ since $t(O) \leq n|\Sigma|$ and $|\mathcal{A}'_O| \geq n$. On top of that, we know that $uav \notin L$ holds for all $v \in \Sigma^*$ because $\bot$ is a nonaccepting sink-state and $\mathcal{A}'_O$ passed an equivalence query. Since $\bot$ has not been merged, there exists no $u' \in R$ with $u' \sim_O ua$. Thus, $\text{index}(\sim_O \cap R \times R) < n$ and, hence, $|\mathcal{A}'_O| \leq n$.

Thus, if $\mathcal{A}'_O$ passes an equivalence query before $\text{index}(\sim_O \cap R \times R) = n$ holds, $\mathcal{A}'_O$ is isomorphic to $\mathcal{A}_L$ because $L(\mathcal{A}'_O) = L$ and $|\mathcal{A}'_O| = n$.

Complexity The observation table can be nonconsistent at most n times. Moreover, $\text{index}(\sim_O \cap R \times R) = n$ and $t(O) = n|\Sigma|$ hold after at most $(n+1)|\Sigma|$ equivalence queries. Thus, Algorithm 7.2 asks $\mathcal{O}(n)$ equivalence queries since we assume Σ to be a priori given and, therefore, constant. Additionally, we can bound the size of the observation table by $\mathcal{O}(mn^2)$ as in Angluin's original algorithm. Therefore, Algorithm 7.2 poses $\mathcal{O}(mn^2)$ membership queries.

Checking for consistency, extending and updating the table, as well as constructing conjectures can be done in time polynomial in the size of the table. Hence, Algorithm 7.2 terminates in time polynomial in m and n if implemented properly. □

Although Algorithm 7.2 explores transitions on demand, it still shows a subtle behavior that is unfavorable in our setting. For instance, consider a situation in which a trace ua with $u \in \Sigma^*$ and $a \in \Sigma$ is added to R and, hence, every future conjecture will contain the transition $\delta_O([\![u]\!]_O, a)$. Assume further that it turns out later in the learning process that some trace $u' \in [\![u]\!]_O$ is not $L(\mathcal{A}_{\mathfrak{G},v_0})$-equivalent to u and both representatives become separated by a separating word. Then, the transition $\delta([\![u]\!]_O, a)$ is kept although we do not know a priori whether the trace u or u' should continue with playing the action a. In the worst case, a strategy automaton that allows continuing the trace u with action a might be considerably larger than one that does not.

This behavior seems to be inherent to Angluin's algorithm and not easy to fix. Therefore, we next develop a variant of Kearns and Vazirani's algorithm that does not suffer from the problem described above. The key idea is to "forget" already discovered transitions every time the algorithm discovers a new state.

A Variant of Kearns and Vazirani's Algorithm

Our variant of Kearns and Vazirani algorithm closely follows the original algorithm as described in Section 3.2.2 (on Page 71 and following) with only a few modifications. As before, our algorithm organizes its data in a classification tree built over two sets $R, S \subseteq \Sigma^*$. The set R consists of representatives used to represent L-equivalence classes, whereas the set S contains separating words that are used to witness that two representatives are not L-equivalent. Moreover, it maintains an auxiliary set $\Delta \subseteq R \times \Sigma$ of *defined transitions* that determines which transitions are present in a conjecture. To avoid the problem of our variant of Angluin's algorithm, we empty Δ every time a new state is discovered.

Constructing conjectures slightly differs from the original algorithm in so far as we direct a transition $\delta_t(u,a)$ to a nonaccepting sink-state if $(u,a) \notin \Delta$. More precisely, from a given classification tree t and a set Δ of defined transitions, we now derive the DFA $\mathcal{A}'_t = (Q_t, \Sigma, q_0^t, \delta_t, F_t)$ where

- $Q_t = R \cup \{\bot\}$;
- $q_0^t = \varepsilon$;
- $F_t = \{u \in R \mid u \cdot \varepsilon \in L\}$ (i.e., all representatives in the root's right subtree); and
- $\delta_t(u,a) = \begin{cases} \text{sift}_t(ua) & \text{if } (u,a) \in \Delta; \\ \bot & \text{otherwise;} \end{cases}$

  and $\delta_t(\bot, a) = \bot$ where $u \in R$ and $a \in \Sigma$.

It might happen that there already exists a $u \in R$ that represents a nonaccepting sink whose outgoing transitions loop back to u, lead to $\bot$, or both. In this case, we merge the states u and $\bot$ in $\mathcal{A}'_t$ analogously to our variant of Angluin's algorithm. Moreover, if a state is not reachable from the initial state, we drop it from $\mathcal{A}'_t$.

As long as $|R| < \text{index}(\sim_L)$, the conjecture $\mathcal{A}'_t$ is necessarily different from $\mathcal{A}_L$ and an equivalence query with $\mathcal{A}'_t$ returns a counterexample. We use this counterexample to either identify a new representative or a missing transition. Therefore, the analysis of a counterexample here is different from the one in Section 3.2.2.

Given a counterexample $w = a_1 a_2 \ldots a_m$, our algorithm searches for the smallest index $i \in \{1, \ldots, m\}$ such that either $\mathcal{A}'_t \colon u_0 \xrightarrow{a_1 \ldots a_{i-1}} u_{i-1}$ and $\delta_t(u_{i-1}, a_i) = \bot$ (i.e., $(u_{i-1}, a_i) \notin \Delta$) or i is a breakpoint position[4]. (We argue shortly that such an index always exists.) In the first case, our algorithm proceeds as the original algorithm and splits the leaf node corresponding to the breakpoint position; moreover, it resets Δ to

[4]Recall that a breakpoint position is an index $i \in \{1, \ldots, m\}$ with $u_{i-1} a_i \ldots a_m \in L \Leftrightarrow u_i a_{i+1} \ldots a_m \notin L$ where u_i satisfies $\mathcal{A}'_t \colon u_0 \xrightarrow{a_1 \ldots a_i} u_i$. Details can be found in Section 3.2.2.

the empty set. In the second case, it adds the pair (u_{i-1}, a_i) to Δ; note that the transition $\delta_t(q_{i-1}, a_i)$ is in fact needed since $\mathcal{A}_t$ rejects the counterexample in this case although it has to be accepted.

Algorithm 7.3 presents our variant of Kearns and Vazirani's algorithm in pseudo code. The initialization phase in Line 1 is the same as in the original algorithm and summarizes Lines 1 to 11 of Algorithm 3.2. Once the algorithm has finished the initialization, it repeats to construct conjectures and process counterexamples until a conjecture passes an equivalence query.

Algorithm 7.3: A variant of Kearns and Vazirani's active learning algorithm.

Input: A teacher for a regular language $L \subseteq \Sigma^*$.

1. Handle the cases $L = \emptyset$ and $L = \Sigma^*$ as in Algorithm 3.2. Return the corresponding DFA if it passes the equivalence query. Otherwise, prepare the initial classification tree t.
2. **repeat**
3. Construct $\mathcal{A}'_t$, and ask an equivalence query with $\mathcal{A}'_t$.
4. **if** *the teacher replies a counterexample* $w = a_1 \dots a_m$ **then**
5. **for** $i = 1$ *to* m **do**
6. Simulate the run $\mathcal{A}'_t$: $u_0 \xrightarrow{a_1} u_1 \xrightarrow{a_2} \dots \xrightarrow{a_i} u_i$.
7. **if** i *is a breakpoint position* **then**
8. $R \leftarrow R \cup \{u_{i-1}a_i\}$, $S \leftarrow S \cup \{a_{i+1} \dots a_m\}$, $\Delta \leftarrow \emptyset$, and update t by splitting the leaf node u_i.
9. **else if** $(u_{i-1}, a_i) \notin \Delta$ **then**
10. $\Delta \leftarrow \Delta \cup \{(u_{i-1}, a_i)\}$.
11. **end**
12. **end**
13. **end**
14. **until** *the teacher returns "yes" on an equivalence query with* $\mathcal{A}'_t$.
15. **return** $\mathcal{A}'_t$.

In summary, we obtain the following result. Again, let us remind the reader that we consider the original active learning setting but not the case that the learning is stopped prematurely.

Theorem 7.9. *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Algorithm 7.3 learns a DFA isomorphic to the canonical minimal DFA $\mathcal{A}_L$ in time polynomial in the size n of $\mathcal{A}_L$ and the length m of the longest counterexample returned by the teacher. It asks $\mathcal{O}(n^2)$ equivalence queries and $\mathcal{O}(mn^3)$ membership queries.*

Proof. Again, we split the proof in two parts: we first show the correctness of Algorithm 7.3 and then estimate the number of queries asked during the learning process.

Let $n = \text{index}(\sim_L)$.

Correctness We first argue that for every counterexample $w = a_1 \dots a_m$, there exists an index $i \in \{1, \dots, m\}$ such that either i is a breakpoint position or $\mathcal{A}'_t \colon u_0 \xrightarrow{a_1 \dots a_{i-1}} u_{i-1}$ and $\delta_t(u_{i-1}, a_i) = \bot$ (i.e., $(u_{i-1}, a_i) \notin \Delta$). To see why, suppose the converse. Then, we have $\mathcal{A}'_t \colon u_0 \xrightarrow{w} u_m$ (since all transitions used in this run are defined). Moreover, since no $i \in \{1, \dots, m\}$ is a breakpoint position, the following holds:

$$\underbrace{u_0}_{\varepsilon} \cdot \underbrace{a_1 \dots a_m}_{w} \in L \Leftrightarrow u_1 \cdot a_2 \dots a_m \in L \Leftrightarrow \dots \Leftrightarrow u_m \cdot \varepsilon \in L.$$

Thus, $w \in L$ if and only if $u_m \in L$. By definition of F_t, we know that $u_m \in F_t$ if and only if $u_m \cdot \varepsilon \in L$. Therefore, $w \in L(\mathcal{A}'_t) \Leftrightarrow w \in L$, which yields a contradiction since w is a counterexample.

Every time a counterexample is processed, Algorithm 7.3 makes progress: it either splits a leaf node and inserts a new representative into R (i.e., $|R|$ increases by one), or it adds a new transition and $|\Delta|$ increases by one. Moreover, we know that $|R| \leq n$ (since representatives are pairwise not L-equivalent) and $|\Delta| \leq n|\Sigma|$ (since any DFA with at most n states over the alphabet Σ contains at most $n|\Sigma|$ transitions). Note, however, that we empty Δ every time a new representative is added to R.

In the worst case, Algorithm 7.3 continues asking equivalence queries until $|R| = n$ and $|\Delta| = n|\Sigma|$. Once these bounds have been reached, we know that all transitions of $\mathcal{A}'_t$ with source $u \in R$ lead to states different from $\bot$. Thus, $\bot$ is unreachable and dropped. This fact allows us to apply the same arguments as in the proof of Theorem 3.10 (on Page 75) to establish that $\mathcal{A}'_t$ is isomorphic to $\mathcal{A}_L$ once $|R| = n$ and $|\Delta| = n|\Sigma|$. Then, $\mathcal{A}'_t$ passes an equivalence query and Algorithm 7.3 terminates.

As with our variant of Angluin's algorithm, it might happen that a conjecture already passes an equivalence query at an earlier point in time. In this case, we obtain that $\mathcal{A}'_t$ is isomorphic to $\mathcal{A}_L$ using arguments similar to those in the proof of Theorem 7.8.

Complexity Algorithm 7.3 asks at most $|R| \cdot |\Sigma|$ equivalence queries until a new representative is added to R. Then, it empties Δ and starts over learning transitions until it extends R again. Thus, Algorithm 7.3 asks $\mathcal{O}(n^2)$ equivalence queries in total because $|R| \leq n$, $|\Delta| \leq n|\Sigma|$, and Σ is assumed to be fixed.

Each equivalence query (except the last) requires a series of membership queries in order to analyze the counterexample. Since Algorithm 7.3 processes counterexamples linearly "from left to right" (but not using a binary search) and the height of a classification tree is bounded by n, it takes $\mathcal{O}(mn)$ membership queries to analyze each counterexample. Moreover, it takes $\mathcal{O}(n^2)$ membership queries to construct a new

conjecture provided that membership queries are cached (the argument is the same as for Kearn's and Vazirani's original algorithm). Since the number of equivalence queries is in $\mathcal{O}(n^2)$, Algorithm 7.3 asks $\mathcal{O}(mn^3)$ membership queries in total.

Finally, let us note that one can perform all operations in time and space polynomial in the size of the classification tree. As with Kearns and Vazirani's original algorithm, it is not hard to verify that the size of a classification tree is linear in n. In addition, we just showed how to process a counterexample in time $\mathcal{O}(mn)$. Thus, Algorithm 7.3 terminates in time polynomial in m and n if implemented properly. □

Note the gap in the number of queries between Algorithm 7.2 and Algorithm 7.3. This difference is caused by the fact that Algorithm 7.3 drops the knowledge about transitions it has already learned. However, this more costly approach helps avoiding the unfavorable behavior of our variant of Angluin's algorithm.

7.2.3 Results of Learning Strategy Automata

We can now prove the correctness of Algorithm 7.1.

Theorem 7.10. *Let $\mathfrak{G} = (\mathfrak{A}, F)$ be a labeled safety game over the Σ-arena $\mathfrak{A}$ and $v_0 \in W_0$. Then, Algorithm 7.1 terminates and returns a strategy automaton $\mathcal{A}$ of size $|\mathcal{A}| \leq |W_0| + 1$ that realizes a winning strategy for Player 0 in $\mathfrak{G}$ from v_0.*

Proof of Theorem 7.10. Let $\mathfrak{G}$ be a labeled safety game, $v_0 \in W_0$ the initial vertex, $\mathcal{A}_{\mathfrak{G},v_0}$ the canonical strategy automaton, and $\mathcal{A}$ the DFA learned by Algorithm 7.1.

The way our teacher for labeled safety games answers queries guarantees that the learning terminates only if the learner conjectures a winning strategy automaton for Player 0 in $\mathfrak{G}$ from vertex v_0. If the learning does not terminate early, the learner learns a strategy automaton that is isomorphic to the minimal DFA accepting $L(\mathcal{A}_{\mathfrak{G},v_0})$. Since we apply learning algorithms that produce conjectures of increasing size, we obtain $|\mathcal{A}| \leq |\mathcal{A}_{\mathfrak{G},v_0}| = |W_0| + 1$. □

The overall time and space complexity of Algorithm 7.1 depends on the choice of the active learning algorithm; for instance, if we plug in our modification of Kearns and Vazirani's algorithm, Algorithm 7.1 asks $\mathcal{O}(mn^3)$ membership queries and $\mathcal{O}(n^2)$ equivalence queries (recall that m is the length of the longest counterexample and n is the size of the minimal DFA accepting the target language). In the present case, the target language is $L(\mathcal{A}_{\mathfrak{G},v_0})$ and, hence, $n \leq |W_0| + 1 \leq |V| + 1$. Moreover, one can bound m by the size $|\mathfrak{A} \times \mathcal{A}|$ of the product of the arena $\mathfrak{A}$ and the conjecture $\mathcal{A}$ because our teacher computes minimal counterexamples. Since we assume that the used learning algorithm conjectures DFAs whose size is at most n, we obtain $m \leq |V|^2$.

Thus, Algorithm 7.1 asks $\mathcal{O}(|V|^4)$ membership queries and $\mathcal{O}(|V|^2)$ equivalence queries in total.

Furthermore, if we assume that the learning algorithm asks membership queries whose length is bounded by m, which is true for both of our modifications, and by considering the runtime of the teacher (see Section 7.2.1), we obtain an overall runtime of Algorithm 7.1 that is polynomial in $|V|$.

Finally, note that Algorithm 7.1 is designed to terminate as early as possible in the learning process, which has an additional beneficial effect on the runtime. In our experiments, which we present next, our prototype indeed learned almost always a strategy automaton that was smaller than the canonical strategy automaton.

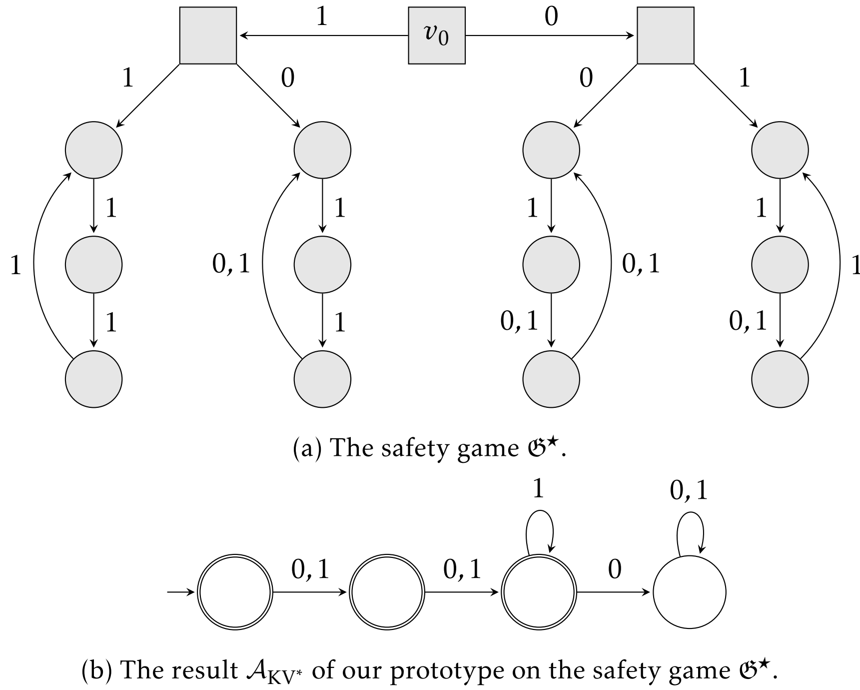
7.3 Experiments and Evaluation

We implemented a C++ prototype of Algorithm 7.1 and the teacher of Section 7.2.1 based on Libalf. This prototype builds upon Kearns and Vazirani's original learning algorithm and implements our modified version thereof. To perform operations on automata, we used the AMoRE++ automata library [BKK+10].

To assess the quality of our prototype, we compared the following five measures:

- The number of vertices of the winning region W_0, denoted by $|W_0|$.
- The size of the strategy automaton learned by our prototype using the original version of Kearns and Vazirani's algorithm, denoted by $|\mathcal{A}_{\mathrm{KV}}|$.
- The size of the strategy automaton learned by our prototype using the modified version of Kearns and Vazirani's algorithm of Section 7.2.2, denoted by $|\mathcal{A}_{\mathrm{KV}^*}|$.
- The size of the minimal DFA accepting $L(\mathcal{A}_{\mathfrak{G},v_0})$, denoted by $|\mathcal{A}^{\min}_{\mathfrak{G},v_0}|$. We obtained $\mathcal{A}^{\min}_{\mathfrak{G},v_0}$ by minimizing the canonical strategy automaton $\mathcal{A}_{\mathfrak{G},v_0}$ using a standard DFA minimization procedure (e.g., see Hopcroft and Ullman [HU79]).
- The size of the minimized implementation of a positional winning strategy f, denoted by $|\mathfrak{A}^{\min}_f|$, which serves as a proxy for tools such as Gavs+ that do not take the size of the resulting implementation into account. To compute positional strategies, we implemented a greedy algorithm[5] that takes the winning region as input and picks for every Player 0 vertex the smallest action (with respect to a given order on the actions) that stays inside the winning region. Once this has been done, we compute the implementation $\mathfrak{A}_f$, interpret it as a DFA, and minimize it.

[5] We decided to fall back on our own implementation because invoking Gavs+ produced a large overhead, which made running the tool too time-consuming.



(a) The safety game $\mathfrak{G}^\star$.

(b) The result $\mathcal{A}_{\mathrm{KV}^*}$ of our prototype on the safety game $\mathfrak{G}^\star$.

Figure 7.5: A safety game and a corresponding winning strategy automaton learned by our prototype.

We conducted all experiments on an Intel Q9550 quad core CPU with 4 GiB of RAM running Ubuntu 12.04 LTS. Our implementation used a single core, and no experiment consumed more than 200 MiB of RAM. Since nearly all experiments finished within less than five minutes, we did not impose any timeout limit.

An Artificial Safety Game

Let us first present an example designed to show that our prototype learns small strategy automata, whereas all other approaches discussed so far produce large solutions.

Consider the $\{0,1\}$-labeled safety game $\mathfrak{G}^\star$ depicted in Figure 7.5a. The idea of this game is as follows: starting in v_0, Player 1 chooses two bits $b_0, b_1 \in \{0,1\}$. Then, it is Player 0's turn, who can win by playing the actions $a_0 = 1$, $a_1 \in \{b_1, 1\}$, and $a_2 \in \{b_2, 1\}$ in this order ad infinitum. Hence, the winning region of Player 0 consists of all vertices. Note that Player 1 can influence whether and when Player 0 can play action 0 with his first moves.

Since Player 1 can decide which part of the arena a play reaches, the automaton $\mathcal{A}_{\mathfrak{G}^\star,v_0}$ covers the whole arena and contains $|V| + 1$ states. Moreover, the game is designed such that $\mathcal{A}_{\mathfrak{G}^\star,v_0}$ and $\mathcal{A}^{\min}_{\mathfrak{G}^\star,v_0}$ coincide because $\mathcal{A}_{\mathfrak{G}^\star,v_0}$ is already minimal. We also observe that the greedy algorithm described above produces a positional strategy,

let us call it f, that plays action 0 whenever possible—we assume the natural order on the alphabet $\{0,1\}$. Thus, both $\mathcal{A}_f$ and $\mathcal{A}_f^{\min}$ contain $|V|+1$ states because the strategy needs to keep track of the exact vertex a finite trace has reached in order to play action 0 whenever possible.

On the other hand, Figure 7.5b depicts the winning strategy automaton $\mathcal{A}_{\mathrm{KV}^*}$ learned by our prototype. This DFA consists of only four states, which is one fourth of the $|V|+1 = 16$ states of $\mathcal{A}^{\min}_{\mathfrak{G}^\star,v_0}$, respectively $\mathfrak{A}_f^{\min}$. Our prototype using Kearns and Vazirani's original algorithm always learned the DFA $\mathcal{A}^{\min}_{\mathfrak{G}^\star,v_0}$.

Let us finally mention that one can generalize the idea of $\mathfrak{G}^\star$: instead of only two bits, Player 1 chooses n bits $b_1,\dots,b_n \in \{0,1\}$, and Player 0 needs to play the actions $a_0 = 1$ and, subsequently, $a_1 \in \{b_1,1\}$ to $a_n \in \{b_n,1\}$ repeatedly in order to win. Again, both $\mathcal{A}^{\min}_{\mathfrak{G}^\star,v_0}$ and $\mathfrak{A}_f^{\min}$ are of size $|V|+1$. On the other hand, experiments up to $n = 20$ showed that our prototype learned strategy automata that have the same structure as shown in Figure 7.5b and comprise $\mathcal{O}(\log|V|)$ states (note that the number of vertices of $\mathfrak{G}^\star$ grows exponentially in n).

Random Safety Games

The lack of a standard benchmark suite for labeled safety games made is necessary to artificially produce examples on which to asses the performance of our prototype. To this end, we implemented a random generator, which produces labeled safety games that structurally resemble systems that arise from composing several subsystems. This generator is parameterized by a number of variables, which influence the shape of the resulting arenas. The remainder of this section presents our experiments on these games.

Methodology Our generator constructs labeled safety games over Σ-arenas with $\Sigma = [m+1]$ where m can be an arbitrary natural number. The exact procedure is a four-step process:

1. The generator creates $c \in \mathbb{N}_+$ *components*, each of which consist of $n \in \mathbb{N}_+$ vertices. All vertices belong to F. A vertex belongs with probability of $p_0 \in [0,1]$ to V_0 and with probability of $1-p_0$ to V_1. Every vertex has exactly one outgoing edge pointing into its own component such that every component is strongly connected. The edges' actions are chosen uniform randomly from Σ.

2. The generator inserts a safe and an unsafe sink-vertex together with self-loops for every action.

3. The generator inserts an edge for every combination of source-vertex and action provided that no such edge already exists. With a probability of $a \in [0,1]$, the

edge leads to a sink—to the unsafe sink if the source of the edge is a Player 0 vertex or to the safe if the source is a Player 1 vertex. If the edge does not lead to a sink, the edge leads with a probability of $b \in [0, 1]$ to a vertex inside another component and with probability $1 - b$ to a vertex inside the same component; in both cases, the destination-vertex is chosen uniform randomly.

4. The generator chooses v_0 uniform randomly from all vertices inside W_0.

We generated two benchmark suites. For the first benchmark suite, we fixed the number of component to $c = 5$ and varied the size n of the components. For the second benchmark suite, we fixed the size of the components to $n = 25$ and varied the number c of components. For each combination of c and n, we generated 1000 games using the parameters $m = 2$ (i.e., $\Sigma = [3]$), $p_0 = 0.5$, $a = 0.1$, and $b = 0.2$. This choice of parameters produced arenas in which most edges point to vertices inside the same component and the winning region of Player 0 covers on average more than half of the arena. The latter fact makes this kind of games particularly interesting as it permits a large number of different winning strategies.

Results Figure 7.6 (on Page 240) presents the results of our experiments; results of the prototype on the first benchmark suite are shown on the left hand side of the figure, whereas the results on the second benchmark suite are shown on the right hand side. The results using Kearns and Vazirani's original algorithm are not shown because the prototype nearly always learned $\mathcal{A}^{\min}_{\mathfrak{G},v_0}$. We averaged the results for each combination of c and n; thus, each data point in Figure 7.6 corresponds to the arithmetic mean of 1000 experiments; by abuse of notation, we denote the arithmetic mean of the values $r_1, \ldots, r_n$ by $\overline{r}$.

Discussion The results on both benchmark suites show a linear dependency between the independent variable (c, respectively n) and the average size of strategy implementations. In both cases, implementations of positional strategies do on average not cover the whole arena but are considerably larger than strategy automata. In the first benchmark suite, implementations of positional strategies are on average four times larger than strategy automata. In the second benchmark suite, strategy automata are of constant size ($\overline{|\mathcal{A}^{\min}_{\mathfrak{G},v_0}|} \approx 27$ and $\overline{|\mathcal{A}_{\mathrm{KV}^*}|} \approx 21.5$), whereas implementations of positional strategies grow with the number of components.

Our prototype using the modified Kearns and Vazirani algorithm nearly always succeeded in learning strategy automata that have less states than the canonical strategy automaton. The average size of $\mathcal{A}_{\mathrm{KV}^*}$ is about 80 % of the average size of $\mathcal{A}^{\min}_{\mathfrak{G},v_0}$ in both benchmark suites. However, Kearns and Vazirani's original algorithm almost always learned the DFA $\mathcal{A}^{\min}_{\mathfrak{G},v_0}$. This shows that running a specialized learning

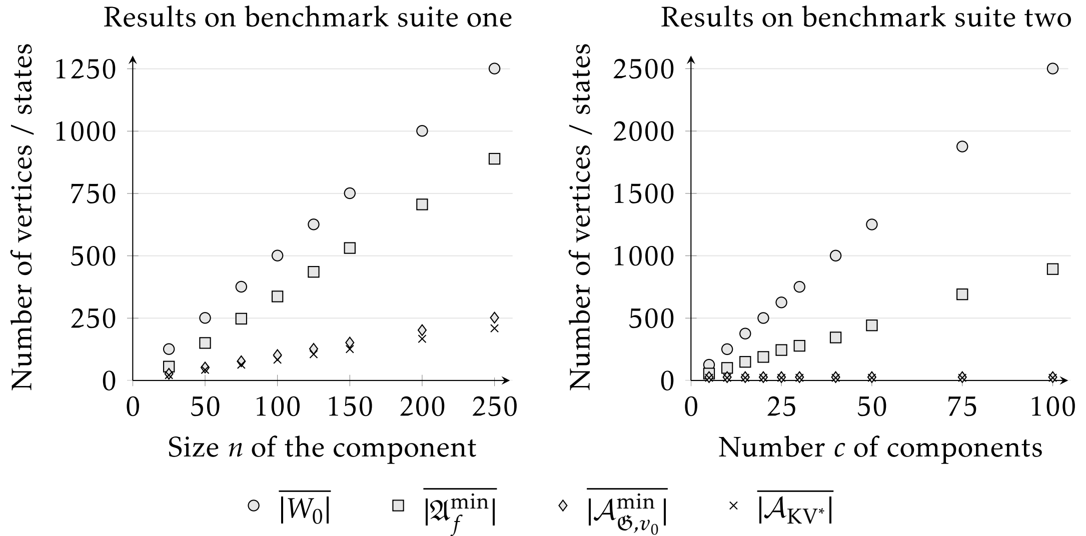


Figure 7.6: Experimental results of our prototype on random safety games.

algorithm on top of just constructing the canonical strategy automaton is indeed advantageous.

To conclude, we observed that automata-based strategies yielded considerably smaller implementations than positional strategies. Furthermore, our prototype nearly always learned winning strategy automata that are smaller than the strategy implementations produced by the other considered techniques.

7.4 Conclusion

We considered labeled safety games, in which the players typically need memory to implement their (winning) strategies. Since computing minimal strategy automata is hard in this setting, we have developed a polynomial time heuristic, which is based on active learning in an Angluin setting. We demonstrated through several experiments with a prototype implementation that our heuristic often produces small winning strategy automata and performs better than classical approaches based on positional strategies. A recent study by Ehlers and Moldovan [EM12] also demonstrated the competitiveness of our approach.

To improve the quality of our results further, we have developed domain-specific modifications of Angluin's algorithm as well as Kearns and Vazirani's algorithm. Both modifications explore transitions on demand rather than by default, as classical active learning algorithms do. Note that this approach might also be helpful in other settings. Consider, for instance, the task of learning DFAs for regular expression, say over the UTF-8 character set. In this situation, the alphabet is large and it can happen

that numerous transitions lead to a nonaccepting sink-state. Thus, probing for all transitions results in a large amount of (unnecessary) membership queries, which one would clearly want to avoid.

A further opportunity for optimizations, which our modifications do not yet exploit, is that the target languages of the present setting are prefix-closed. However, besides straightforward filter operations on membership queries (e.g., not posing membership queries for prefixes of words already known to be accepting), it is not clear how an intelligent learning algorithm for prefix-closed languages should be designed. Nonetheless, prefix-closed languages are not restricted to the present setting and occur in other areas as well. Thus, an optimized learning algorithm would be of general interest.

Finally, it would be interesting to investigate how the idea of learning winning strategies can be lifted to games with more complex winning condition, such as Büchi, Muller, or parity conditions. Although a technique to "reduce" Muller games to safety games has recently been presented [NRZ12], which in turn allows applying our techniques to Büchi and parity games as well, the safety game resulting from this "reduction" can be exponential in the size of the given arena. Thus, a direct approach seems worthwhile, although it entails the challenge of finding a suitable representation of winning strategies. A likely candidate might be the class of ω-regular languages, for which Angluin-style learning algorithms exist (e.g., developed by Maler and Pnueli [MP95] as well as Farzan et al. [FCC+08]).

8

Conclusion

The objective of this thesis has been to lift the merits of automata learning techniques to the areas of verification and synthesis. In this context, we explored four application scenarios that have turned out to be particularly well-suited for automata learning: Regular Model Checking, quantified invariants of linear data structures, automatic reachability games, and labeled safety games.

Regular Model Checking We have developed three different kinds of algorithms:

- A white-box algorithm that does not build upon automata learning
- Learning-based semi-black-box algorithms that use active learning to abstract from the sets of initial and bad configurations (but still need access to the transducer of the program)
- Learning-based black-box algorithms that obtain their information exclusively from a teacher (who reasons about the program) and, thus, can also be used if the program is not given in terms of finite automata (as long as one can construct an appropriate teacher)

The black-box algorithms are based on the ICE-learning framework, which has recently been introduced as a general concept for invariant synthesis.

All of these algorithms share the same underlying idea, which is to delegate intricate and costly calculations to off-the-shelf SAT and SMT solvers. For the semi-black-box and the black-box algorithms, we have developed two types of learners: the first type follows the CEGAR principle and progressively refines its abstraction of the program by means of counterexamples; the second type extends this idea and additionally leverages Angluin's algorithm to the present setting.

To put our algorithms into context, we implemented a prototype and compared its performance to the established tools Faster and T(o)rmc. Moreover, we applied our algorithms to two alternative scenarios, namely the problem of finding minimal separating DFAs and the problem of synthesizing Presburger loop invariants for While programs.

Quantified invariants of linear data structures We studied the problem of automatically synthesizing universally quantified invariants for programs manipulating linear data structures such as arrays and lists. To capture such invariants, we modeled arrays and lists as data words, respectively valuation words. Building upon these notions, we have introduced a novel automaton model, called quantified data automata, and have developed an active learning algorithm for this type of automata that reduces the learning task to the task of learning Moore machines.

Since quantified data automata can express undecidable properties of data structures, we have identified a syntactic restriction called elastic quantified data automata. These kind of automata still allow expressing interesting properties and can be translated into the Array Property Fragment (in the case of arrays) or the decidable syntactic fragment of Strand (in the case of lists), which are both decidable logics. Moreover, we have developed a procedure, called elastification, which makes it possible to uniquely abstract from a quantified data automaton to an elastic quantified data automaton.

We embedded our active learning algorithm in a passive learning setting. To this end, we constructed a teacher who answers queries with respect to a finite sample consisting of "short" data structures (over a "small", finite abstraction of the data domain) that manifest during program executions. The motivation for this approach is that invariants typically neither depend on the length of an array (or list) nor on the actual data contained therein and, thus, a small sample of the observed program behavior is often sufficient to identify an invariant. To demonstrate the effectiveness of this approach, we implemented a prototype and successfully used it to synthesize invariants for numerous real-world programs, including device drivers and parts of the GNU core utilities.

Automatic reachability games In order to study automata learning in the context of infinite games, we have introduced automatic reachability games, which are classical reachability games played on arenas whose underlying graphs are automatic.

As a first step, we considered automatic reachability games over finite arenas and have developed a DFA-based algorithm for computing attractors. This algorithm follows the same scheme as the classical attractor computation but performs all operations symbolically by means of DFAs.

In order to solve automatic reachability games over infinite arenas, we have developed an learning-based algorithm working in an active learning setting that does not learn the attractor directly but rather the unique fixed point of a certain functional. This functional allows both constructing a teacher (which uses parts of the algorithm for automatic reachability games over finite arenas) and applying standard active learning algorithms.

We implemented a prototype and demonstrated the competitiveness of the DFA-based attractor computation using a benchmark suite introduced by Alur, Madhusudan, and Nam. Moreover, we extended this suite with a game over an infinite arena in order to demonstrate the performance of the learning-based algorithm.

Labeled safety games Finally, we considered the task of computing small implementations of strategies, a task on which not much research has been spent so far. As concrete scenario, we considered labeled safety games, which are safety games played on finite arenas whose underlying graphs are deterministically labeled with actions.

Since computing small implementations of strategies is computationally hard in this context, we have developed a heuristic that exploits a common property of active learning algorithms, namely that such algorithms produce conjectures of increasing size. The heuristic builds upon the notion of (winning) strategy automata, which we have introduced as a means to represent (winning) strategies. The key idea of our heuristic is to prematurely stop the learning as soon as the learner conjectures a winning strategy automaton.

As optimizations, we have developed modifications of Angluin's as well as Kearns and Vazirani's learning algorithms, which probe for transitions only when necessary. Using a prototype implementation, we demonstrated that learning with these modified algorithms indeed yields smaller implementations of strategies than presently existing tools, which derive positional winning strategies in a greedy manner.

In conclusion, this thesis proves that automata learning is a powerful tool: not only have we developed competitive learning-based algorithms for various application scenarios, but also used automata learning to advance the state of the art in the areas of verification and synthesis (e.g., ICE-learning, which we have introduced as a novel, generic approach to invariant generation).

However, automata learning is not a one-size-fits-all solution. In particular, automata learning relies crucially on a well-suited representation of the considered problem (usually in terms of regular languages), which makes a careful analysis and design imperative. An unsuitable encoding, on the other hand, quickly leads to a poor performance of the learning algorithms and, hence, of the overall algorithm. These insights are reflected in the following list of future research.

Open Questions and Future Research

We already discussed future research in the conclusions of Chapters 4 to 7. Here, we summarize some open questions as a guide to future research.

Our algorithms for Regular Model Checking and for solving automatic reachability games are necessarily incomplete due to the properties of the underlying graphs (which, in turn, are determined by the underlying transducers). Despite the fact that (a)synchronous transducers can encode computations of Turing machines, it is not well understood what kind of (syntactic) properties make a transducer define a "complex" graph. In connection with automata learning, this gives rise to the following question.

Question 1. *What are nontrivial syntactic properties of a transducer that guarantee regularity of the target language?*

A major contribution of this thesis has been to lift automata learning to the area of infinite games. In particular, we studied automata learning in the context of (automatic) reachability games and (labeled) safety games. However, we did not yet consider other winning conditions, such as Büchi, Muller, or parity conditions. This immediately leads to the following question.

Question 2. *Can automata learning be applied to the whole class (or at least to an "interesting" subclass) of ω-regular winning conditions?*

One can raise a similar question regarding the target concept of learning algorithms. Since all techniques that we have devised in the present thesis use DFAs as the only target concept, a natural question is whether the class of deterministic finite automata is the only tractable target concept.

Question 3. *To what extent can one lift the techniques developed in this thesis to more expressive automata models?*

An answer to this question clearly entails the question of what kind of formal languages are learnable in general (either actively or passively).

By utilizing the ICE-learning framework, we were able to introduce an active learning setting for invariant generation in which a teacher can communicate all information necessary to learn an (unknown) invariant. However, the fact that ICE-learning is a generic concept gives rise to the question of where else ICE-learning can be used.

Question 4. *To what (verification) domains can ICE-learning be applied?*

We addressed this question to some extent by considering template-based ICE-learning for numerical constraints and learning of quantified invariants of linear data structures that we studied in Chapter 5 [GLMN13a, GLMN14] (the latter still being an

elaborate manual task on a per-program basis). Moreover, it would be helpful to identify properties a domain needs to exhibit in order to allow for the application of ICE-learning.

A follow-up question is whether the ICE-learning framework can also be used to directly learn attractors of automatic reachability games (i.e., without making a detour via learning the fixed point of an auxiliary functional). More generally, it would be interesting to answer the following question.

Question 5. *Can one lift ICE-learning to the area of infinite games?*

This list of open questions shows that there is much interesting work to be done in the area of automata learning, in particular in the context of verification and synthesis.

Bibliography

[ABG+12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Safari: Smt-based abstraction for arrays with interpolants. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 679–685. Springer, 2012.

[ACMN05] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 98–109. ACM, 2005.

[AHM+98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 1998.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 555–568. Springer, 2002.

[AJNS04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[AMN05a] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[AMN05b] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *STTT*, 7(2):118–128, 2005.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[ARMS03] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In Craig Boutilier, editor, *IJCAI*, pages 399–404, 2009.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[BDdM+13] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 years of smt-comp. *J. Autom. Reasoning*, 50(3):243–277, 2013.

[BDES12] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2012.

[BDGW97] 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, 1997.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[BF09] Patricia Bouyer and Vojtech Forejt. Reachability in stochastic timed games. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *ICALP (2)*, volume 5556 of *Lecture Notes in Computer Science*, pages 103–114. Springer, 2009.

[BFL04] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. Faster acceleration of counter automata in practice. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590. Springer, 2004.

[BFLP03] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast: Fast acceleration of symbolikc transition systems. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BG04] Achim Blumensath and Erich Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory Comput. Syst.*, 37(6):641–674, 2004.

[BGG01] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Universitext. Springer, 2001.

[BGJ+07] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[BHKL09] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In Craig Boutilier, editor, *IJCAI*, pages 1004–1009, 2009.

[BHRV12] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular (tree) model checking. *STTT*, 14(2):167–191, 2012.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie97] Armin Biere. μcke - efficient μ-calculus model checking. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer, 1997.

[Bie08] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[BJK10] Tomás Brázdil, Petr Jancar, and Antonín Kucera. Reachability games on extended vector addition systems with states. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 478–489. Springer, 2010.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKK+10] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer, 2010.

[BLP06] Sébastien Bardin, Jérôme Leroux, and Gérald Point. Fast extended release. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66. Springer, 2006.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2003.

[BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 561–575. Springer, 2004.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bro11] Václav Brozek. Optimal strategies in infinite-state stochastic reachability games. In Giovanna D'Agostino and Salvatore La Torre, editors, *GandALF*, volume 54 of *EPTCS*, pages 60–73, 2011.

[Cac02] Thierry Cachat. Symbolic strategy synthesis for games on pushdown graphs. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 704–715. Springer, 2002.

[CFC+09] 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, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2009.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[Chu57] Alonzo Church. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, 1:3–50, 1957.

[CKLB11] Chih-Hong Cheng, Alois Knoll, Michael Luttenberger, and Christian Buckl. Gavs+: An open platform for the research of algorithmic game solving. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 258–261. Springer, 2011.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *STOC*, pages 151–158. ACM, 1971.

[CP05] Jean-Marc Champarnaud and Thomas Paranthoën. Random generation of dfas. *Theor. Comput. Sci.*, 330(2):221–235, 2005.

[CW12] Yu-Fang Chen and Bow-Yaw Wang. Learning boolean functions incrementally. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2012.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *ICSE*, pages 411–420. ACM, 1999.

[dAHK07] Luca de Alfaro, Thomas A. Henzinger, and Orna Kupferman. Concurrent reachability games. *Theor. Comput. Sci.*, 386(3):188–217, 2007.

[DJW97] Stefan Dziembowski, Marcin Jurdzinski, and Igor Walukiewicz. How much memory is needed to win infinite games? In *LICS*, pages 99–110. IEEE Computer Society, 1997.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[dlH05] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[DLS01] Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2001.

[DLT04] François Denis, Aurélien Lemay, and Alain Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dMB09] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2009.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *ICSE*, pages 449–458. ACM, 2000.

[Ehl11] Ruediger Ehlers. Small witnesses, accepting lassos and winning strategies in omega-automata and games. *CoRR*, abs/1108.0315, 2011.

[ELS11] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning workflow petri nets. *Fundam. Inform.*, 113(3-4):205–228, 2011.

[EM12] Rüdiger Ehlers and Daniela Moldovan. Sparse positional strategies for safety games. In Doron Peled and Sven Schewe, editors, *SYNT*, volume 84 of *EPTCS*, pages 1–16, 2012.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FCC+08] 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, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008.

[FH10] Nathanaël Fijalkow and Florian Horn. The surprizing complexity of reachability games. *CoRR*, abs/1010.2420, 2010.

[FKP10] Lu Feng, Marta Z. Kwiatkowska, and David Parker. Compositional verification of probabilistic systems using learning. In *QEST*, pages 133–142. IEEE Computer Society, 2010.

[FKP11] Lu Feng, Marta Z. Kwiatkowska, and David Parker. Automated learning of probabilistic assumptions for compositional reasoning. In Dimitra Giannakopoulou and Fernando Orejas, editors, *FASE*, volume 6603 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2011.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[FL02] Alain Finkel and Jérôme Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In Manindra Agrawal and Anil Seth, editors, *FSTTCS*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2245 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2001.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[FR95] Amr F. Fahmy and Robert S. Roos. Efficient learning of real time one-counter automata. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *ALT*, volume 997 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 1995.

[Gel12] Marcus Gelderie. Strategy machines and their complexity. In Branislav Rovan, Vladimiro Sassone, and Peter Widmayer, editors, *MFCS*, volume 7464 of *Lecture Notes in Computer Science*, pages 431–442. Springer, 2012.

[GH11] Marcus Gelderie and Michael Holtmann. Memory reduction via delayed simulation. In Johannes Reich and Bernd Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 46–60, 2011.

[GLMN13a] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust learning framework for synthesizing invariants. Technical report, RWTH Aachen University and University of Illinois at Urbana-Champaign, 2013.

[GLMN13b] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 813–829. Springer, 2013.

[GLMN13c] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. *CoRR*, abs/1302.2273, 2013.

[GLMN14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[GLP06] Olga Grinchtein, Martin Leucker, and Nir Piterman. Inferring network invariants automatically. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2006.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[HD05] Paul Hunter and Anuj Dawar. Complexity bounds for regular games. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *MFCS*, volume 3618 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2005.

[HJJ+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[HL07] Michael Holtmann and Christof Löding. Memory reduction for strategies in infinite games. In Jan Holub and Jan Zdárek, editors, *CIAA*, volume 4783 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2007.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley series in computer science. Addison-Wesley, 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley series in computer science. Addison-Wesley, 1979.

[HV05] Peter Habermehl and Tomás Vojnar. Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.*, 138(3):21–36, 2005.

[HV10] Marijn Heule and Sicco Verwer. Exact dfa identification using sat solvers. In José M. Sempere and Pedro García, editors, *ICGI*, volume 6339 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2010.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2007.

[Ker09] Carsten Kern. *Learning Communicating and Nondeterministic Automata*. PhD thesis, RWTH Aachen University, August 2009.

[KJD+10] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In Kazunori Ueda, editor, *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2010.

[KLVY11] Orna Kupferman, Yoad Lustig, Moshe Y. Vardi, and Mihalis Yannakakis. Temporal synthesis for bounded systems and environments. In Thomas Schwentick and Christoph Dürr, editors, *STACS*, volume 9 of *LIPIcs*, pages 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[KMM+97] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich ssertional languages. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 1997.

[KMP+89] V. Kell, Albert Maier, Andreas Potthoff, Wolfgang Thomas, and U. Wermuth. Amore: A system for computing automata, monoidsm and regular expressions. In Burkhard Monien and Robert Cori, editors, *STACS*, volume 349 of *Lecture Notes in Computer Science*, pages 537–538. Springer, 1989.

[Koh70] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1970.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. Springer, 1 edition, 2008.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *Intro Computational Learning Theory*. Mit Press, 1994.

[Leg07] Axel Legay. *Generic Techniques for the Verification of Infinite-State Systems*. PhD thesis, Université de Liège, December 2007.

[Leg08] Axel Legay. T(o)rmc: A tool for (omega)-regular model checking. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 548–551. Springer, 2008.

[Ler05] Jérôme Leroux. A polynomial time presburger criterion and synthesis for number decision diagrams. In *LICS*, pages 147–156. IEEE Computer Society, 2005.

[Leu06] Martin Leucker. Learning meets verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2006.

[LN12] Martin Leucker and Daniel Neider. Learning minimal deterministic automata from inexperienced teachers. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2012.

[LPP98] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki,

editors, *ICGI*, volume 1433 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[McM02] Kenneth L. McMillan. Applying sat methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM08] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2008.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149 – 184, 1993.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[Mø10] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[MP95] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2):316–326, 1995.

[MPQ11] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 611–622. ACM, 2011.

[MPX+13] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In Vivek Sarkar and Rastislav Bodík, editors, *ASPLOS*, pages 293–304. ACM, 2013.

[MQ11] P. Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using strand. In Eran Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2011.

[MSHM11] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.

[Nei10] Daniel Neider. Reachability games on automatic graphs. In Michael Domaratzki and Kai Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 222–230. Springer, 2010.

[Nei11] Daniel Neider. Small strategies for safety games. In Tevfik Bultan and Pao-Ann Hsiung, editors, *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2011.

[Nei12] Daniel Neider. Computing minimal separating DFAs and regular invariants using SAT and SMT solvers. In Supratik Chakraborty and Madhavan Mukund, editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2012.

[NJ13] Daniel Neider and Nils Jansen. Regular model checking using solver technologies and automata learning. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2013.

[NL10] Daniel Neider and Christof Löding. Learning visibly one-counter automata in polynomial time. Technical Report AIB-2010-02, Lehrstuhl Informatik 7, RWTH Aachen University, January 2010.

[NRZ12] Daniel Neider, Roman Rabinovich, and Martin Zimmermann. Down the borel hierarchy: Solving muller games via safety games. In Marco Faella and Aniello Murano, editors, *GandALF*, volume 96 of *EPTCS*, pages 169–182, 2012.

[OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[Opp78] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, 1978.

[OS98] Arlindo L. Oliveira and João P. Marques Silva. Efficient search techniques for the inference of minimum size finite automata. In *SPIRE*, pages 81–89, 1998.

[OS01] Arlindo L. Oliveira and João P. Marques Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44(1/2):93–119, 2001.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[Pfl73] C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Trans. Comput.*, 22(12):1099–1102, December 1973.

[PO98] Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *ICCAD*, pages 482–489, 1998.

[PP04] Dominique Perrin and Jean-Éric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, first edition, 2004.

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[Sch91] Robert Elias Schapire. *The Design and Analysis of Efficient Learning Algorithms*. PhD thesis, Massachusetts Institute of Technology, February 1991.

[SG09] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 223–234. ACM, 2009.

[SNA12] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2012.

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2009.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[TB73] Boris A. Trakhtenbrot and Ian M. Barzdin. *Finite Automata; Behavior and Synthesis*. Sequential Machine Theory. North-Holland Publishing Company; New York: American Elsevie, 1973.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

[Tho01] Wolfgang Thomas. A short introduction to infinite automata. In Werner Kuich, Grzegorz Rozenberg, and Arto Salomaa, editors, *Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2001.

[Tho08a] Wolfgang Thomas. Church's problem and a tour through automata theory. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science*, volume 4800 of *Lecture Notes in Computer Science*, pages 635–655. Springer, 2008.

[Tho08b] Wolfgang Thomas. Solution of Church's problem: A tutorial. In K. Apt and R. van Rooij, editors, *New Perspectives on Games and interaction*, volume 4. Amsterdam University Press, 2008.

[Tho09] Wolfgang Thomas. Facets of synthesis: Revisiting church's problem. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009.

[Tou01] Tayssir Touili. Regular model checking using widening techniques. *Electr. Notes Theor. Comput. Sci.*, 50(4):342–356, 2001.

[Val84] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[VSVA04a] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Actively learning to verify safety for fifo automata. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 494–505. Springer, 2004.

[VSVA04b] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning to verify safety properties. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.

[VSVA05] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Using language inference to verify omega-regular properties. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2005.

[VV05] Abhay Vardhan and Mahesh Viswanathan. Learning to verify branching time properties. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 325–328. ACM, 2005.

[VV06] Abhay Vardhan and Mahesh Viswanathan. Lever: A tool for learning based verification. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 471–474. Springer, 2006.

Index

$\mathcal{A}_{\mathfrak{G},v_0}$, 215
$\mathfrak{A}_f$, 214
$\mathcal{A}_L$, 15
$\mathcal{A}^{\varphi}$, 27

Action, 210
Active learning, 63
Algorithmic learning of finite automata, 41
Alphabet, 12
  Input, 15
  Output, 15
Angluin's algorithm, 67
Angluin-style ICE-learner, 122
Angluin-style learner, 111
Arena, 34
  automatic, 183
  finite, 34
  infinite, 34
  Σ-arena, 210
Arity, 20
Array Property Fragment, 29
Attractor, 38
Automata learning, *see* Algorithmic learning of finite automata

Büchi game, 35
Black-box, 81
Breakpoint position, 72

Canonical minimal DFA, 14
Canonical order on words, 13
Cartesian lattice, 144
CEGAR principle, 85, 103
CEGAR-style ICE-learner, 120
CEGAR-style learner, 108
Church's synthesis problem, 7
Classification tree, 71
Clause, 25
Concatenation
  of languages, 12
  of words, 12
Configuration
  bad, 90
  initial, 90
Congruence, 14
Conjunctive normal form, 25
Constant, 20
Convolution, 15
Counterexample, 63

Data formula, 144
Data word, 142
DFA, *see* finite automaton, deterministic
Domain of a sort, 21

Edge, 18
Elastification, 153
Element logic, 29
Empty word, 12
Encoding of pairs, 190
EQDA, *see* Quantified data automaton, elastic
Equality Logic, 25
Equivalence class, 12
Equivalence query, 62
Equivalence relation, 12
Extended observation table, 110
  weakly closed, 111

- weakly consistent, 111

Finite automaton
- deterministic, 13
- inductive, 93
- minimal, 14
- nondeterministic, 13

Formula
- atomic, 21
- equivalent formulas, 22
- first-order, 21
- quantifier free, 21
- solving, 23

Formula word, 149
Function
- Moore machine-computable, 14

Function symbol, 20

Game, 35
- determined, 37
- State, 198

Graph, 18
- automatic, 18
- finite, 18
- infinite, 18
- rational, 18

ICE, *see* ICE-learning
ICE-learning, 113
ICE-teacher, 114
IDFA, *see* Invariant-DFA
Image, 11
Implication counterexample, 86
Index, 12
Index guard, 31
Index logic, 29
Inductive set, 92
Interpretation, 21
Invariant, 92
Invariant-DFA, 93
Irrelevant self-loop, 158

Kearns and Vazirani's algorithm, 73

Language, 12
- prefix-closed, 12

Learner, 62
Literal, 24
Loop invariant, 130

Membership query, 62
Model, 22
Monotonicity, 39, 189
Moore machine, 14
Muller game, 36
Myhill-Nerode congruence, 14

Nerode congruence, *see* Myhill-Nerode congruence
NFA, *see* finite automaton, nondeterministic

Observation table, 64
- closed, 64
- consistent, 64
- reduced, 77

Obviously different words, 47

P_A, 158
Parity game, 36
Passive learning, 43
Play, 34
- consistent with strategy, 37

Player, 34
Postcondition, 130
Postfix, 12
Precondition, 130
Predicate symbol, 20
Prefix, 12
Prefix acceptor, 43
Prefix point, 192
Preimage, 11
Presburger arithmetic, 26
Program, 90
Propositional Boolean Logic, 24
Pursuit-evasion game, 197

QDA, *see* Quantified data automaton
Quantified data automaton, 145
- elastic, 152

Reachability game, 35
  automatic, 184
Reflexive and transitive closure, 12
Regular Model Checking, 90
Relation
  automatic, 16
  binary, 11
  Inverse, 11
  rational, 16
  reflexive and transitive closure, 12
  transitive closure, 11
Representative, 64, 71

Safety game, 35
  classical, *see* Safety game
  Σ-labeled, 211
Sample, 42
SAT, *see* Propositional Boolean Logic
Satisfiability problem, 23
Sentence, 21
Separating DFA, 128
Separating word, 64, 71
Sifting, 72
Signature, 20
Simple path, 157
Size of a strategy implementation, 216
Solver, 23
  SAT, 24
  SMT, 26
Sort, 20
Strand, 32
  decidable syntactic fragment, 33
Strategy, 36
  positional, 37
    Implementation, 214
  winning, 37
    uniform, 37
Strategy automaton, 214
  canonical, 215
  size, 216
  winning, 214
Subformula, 21
Swap game, 197
Symbol, 12
Symbolic word, 143
Synthesis, 7

Target language, 62
Target vertices, 184
Teacher, 62
  for labeled safety games, 223
  ICE, *see* ICE-teacher
  incomplete, 102
Term, 20
Trace, 211
  finite, 211
Trace strategy, 212
Transducer
  asynchronous, 16
  synchronous, 15
Transitive closure, 11

Unbounded swap game, 198
Uninterpreted function, 28
Uninterpreted predicate, 28

Valuation, 143
Valuation word, 143
Value constraint, 31
Variable, 20
  bound, 21
  free, 21
Verification, 3
Vertex, 18
  safe, 211

While loop, 130
While program, 129
White-box, 81
Winning condition, 35
  ω-regular, 36
Winning region, 37
Word, 12
  convoluted, 15
  infinite, 13
  length, 12
  negatively classified, 42
  positively classified, 42