RWTH Aachen

Department of Computer Science
Technical Report

Learning Communicating and Nondeterministic Automata

Carsten Kern

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-17

RWTH Aachen · Department of Computer Science · September 2009

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Learning Communicating and Nondeterministic Automata

Von der Fakultät für Mathematik, Informatik und
Naturwissenschaften der Rheinisch-Westfälischen
Technischen Hochschule Aachen zur Erlangung des
akademischen Grades eines Doktors der
Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Carsten Kern
aus
Aachen

Berichter: Prof. Dr. Ir. Joost-Pieter Katoen
Prof. Dr. Bengt Jonsson

Tag der mündlichen Prüfung: 31. August 2009

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Carsten Kern
Lehrstuhl Informatik 2
`kern@cs.rwth-aachen.de`

Aachener Informatik-Bericht AIB-2009-17

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935–3232

Abstract

The results of this dissertation are two-fold. On the one hand, inductive learning techniques are extended and two new inference algorithms for inferring nondeterministic, and universal, respectively, finite-state automata are presented. On the other hand, certain learning techniques are employed and enhanced to semi-automatically infer communicating automata (also called *design models* in the software development cycle). For both topics, theoretical results on the feasibility of the approaches, as well as an implementation are presented which, in both cases, support our theory.

Concerning the first objective to derive a so-called *active online learning algorithm* for nondeterministic finite-state automata (NFA), we present, in analogy to Angluin's famous learning algorithm L^* [Ang87a] for deterministic finite-state automata (DFA), a version for inferring a certain subclass of NFA. The automata from this class are called *residual finite-state automata* (RFSA). It was shown by Denis et al. [DLT02] that there is an exponential gap between the size of minimal DFA and their corresponding minimal RFSA. Even if there are also cases where the canonical (i.e., minimal) RFSA is exponentially larger than a corresponding minimal NFA, we show that the new learning algorithm—called NL^*—is a great improvement compared to L^* as the inferred canonical RFSA has always at most the size of the corresponding minimal DFA but is usually even considerably smaller and more easy to learn. Unlike a learning procedure developed by Denis et al.—called DeLeTe2 [DLT04]—our algorithm is capable of deriving canonical RFSA. Like L^*, the new algorithm will be applicable in many fields including pattern recognition, computational linguistics and biology, speech recognition, and verification. From our point of view, NL^* might especially play a mayor role in the area of formal verification where the size of the models that are processed is of enormous importance and nondeterminism not regarded an unpleasant property.

The second objective of this thesis is to create a method for inferring distributed design models (CFMs) from a given set of requirements specified as scenarios (message sequence charts). The main idea is to extend the L^* algorithm to cope with valid and invalid sets of system runs and, after some iterations, come up with an intermediate design model (a DFA) which exhibits features that make it distributable into communicating components (or processes) interacting via FIFO channels. Theoretical results on which classes of CFMs are learnable in which time-complexity bounds are presented. We also developed a tool implementation called `Smyle`, realizing important theoretical results evolving from this part of the thesis. Based on this learning formalism we also derive a software-engineering lifecycle model called the `Smyle` *Modeling Approach* in which we embedded our learning approach.

Additionally, we launched a project for a new learning library called `libalf` which includes most of the learning algorithms (and their extensions) mentioned in this thesis. We hope that, due to its continuously increasing functionality, `libalf` will find broad acceptance among researchers, and that it will be the starting point for an extensive project of different research groups which will employ `libalf`, or augment the library with new algorithms.

Zusammenfassung

Die Ergebnisse dieser Arbeit sind zweigeteilt. Einerseits werden existierende induktive Lernverfahren grundlegend erweitert, und zwei neue Lernalgorithmen zur Inferenz nichtdeterministischer bzw. universeller Automaten vorgestellt. Andererseits werden bestimmte Lerntechniken eingesetzt und verbessert, um kommunizierende Automaten (in der Softwareentwicklung auch *Designmodelle* genannt) semi-automatisch zu inferieren. Es werden sowohl theoretische Resultate bezüglich der Durchführbarkeit der Ansätze als auch zwei Implementierungen vorgestellt, die jeweils unsere theoretische Arbeit untermaueren.

Für das erste Ziel, nämlich einen sogenannten *aktiven online Lernalgorithms* für nichtdeterministische, endliche Automaten (NFA) zu entwerfen, wird in Analogie zu Angluins berühmtem Lernalgorithmus L* [Ang87a] für determinisitische, endliche Automaten (DFA) eine Version eines Algorithmus entwickelt, der eine bestimmte Teilklasse nichtdeterministischer, endlicher Automaten inferieren kann. Die Automaten dieser Klasse werden residuelle, endliche Automaten (RFSA) genannt. Denis et al. haben in [DLT02] gezeigt, dass es eine exponentielle Kluft zwischen der Größe minimaler DFA und äquivalenter kanonischer RFSA gibt. Auch in Fällen in denen der kanonische RFSA exponentiell größer ist als ein äquivalenter minimaler NFA, zeigen wir, dass unser neuer Lernalgorithmus NL* eine große Verbesserung im Vergleich zu dem schon existierenden Algorithmus L* darstellt, da der kanonische RFSA im schlimmsten Fall die gleiche Größe wie der äquivalente minimale DFA besitzt, im Regelfall jedoch wesentlich kleiner und leichter zu lernen ist. Im Gegensatz zu dem von Denis et al. in [DLT04] vorgestellten Lernverfahren ist unser Algorithmus in der Lage kanonische RFSA zu lernen. Wie L* so wird auch unser Algorithmus NL* in vielen Bereichen Einsatz finden: z.B. in der Mustererkennung, in der maschinellen Linguistik und Biologie sowie in der Spracherkennung und formalen Verifikation. Unserer Meinung nach könnte NL* vor allem im Bereich der formalen Verifikation eine Hauptrolle spielen, in der die Größe der zu prüfenden Modelle von essentieller Bedeutung und Nichtdeterminismus nur von untergeordnetem Interesse ist.

Das zweite Ziel dieser Arbeit lautet, einen Ansatz zur Inferenz verteilter Designmodelle (CFMs) basierend auf Anforderungen zu kreieren, die als Systemszenarios (Message Sequence Charts) spezifiziert sind. Die Hauptidee besteht darin, den Algorithmus von Angluin dahingehend zu erweitern, dass er mit gültigen und ungültigen Systemläufen umzugehen weiß und nach einigen Iterationen mit einem Zwischenmodell (einem DFA) aufwartet, das Eigenschaften aufweist, die es in über FIFO Kanäle kommunizierende Komponenten (oder Prozesse) verteilen lassen. Es werden theoretische Ergebnisse hergeleitet, die Aussagen darüber liefern, welche Klassen verteiler Automaten (CFMs) mit welchen Komplexitätsschranken lernbar sind. In diesem Zusammenhang wird außerdem eine Implementierung mit Namen `Smyle` vorgestellt, die die Durchführbarkeit des Lernens einiger in dieser Arbeit vorgestellter Klassen aufzeigt. Aufbauend auf diesem theoretischen Lernformalismus wird ein softwaretechnisches Lebenszyklusmodell, genannt der `Smyle` *Modeling Approach*, entwickelt, in das unsere Lernanwendung `Smyle` integriert wird.

Zusätzlich wurde ein Projekt ins Leben gerufen, das aktuell die meisten der in dieser Arbeit vorgestellten Lernverfahren und deren Erweiterungen implementiert. Die entstandene Lernbibliothek trägt den Namen `libalf`. Wir hoffen, dass diese Bibliothek aufgrund ihres kontinuierlich ansteigenden Umfangs an Lernverfahren großen Anklang unter Wissenschaftlern finden wird, und dass sie der Startpunkt eines weitläufigen Projektes unterschiedlicher Universitäten wird, die diese Bibliothek nutzen und dazu beitragen, sie mit neuen Algorithmen an zu reichern.

Acknowledgments

My first and utmost thanks go to my supervisor Professor Dr. Ir. Joost-Pieter Katoen for being, at all times, a pleasant, motivating, and challenging employer, colleague, and mentor, and always encouraging me to present research results at international conferences.

I am equally beholden to my former diploma thesis supervisor, current coauthor, and friend Dr. Benedikt Bollig who always elated me for starting and continuing the challenge of doing a PhD, even when I was facing difficult situations.

I am grateful to Professor Dr. Bengt Jonsson for immediately agreeing on being the second supervisor for this dissertation.

Moreover, I would like to unendingly thank my coauthors Professor Dr. Ir. Joost-Pieter Katoen, Dr. Benedikt Bollig, Professor Dr. Martin Leucker, and Dr. Peter Habermehl for sharing their knowledge and expertise with me, for the great, pleasant but at the same time professional and inspiring research atmosphere, for interesting discussions, and for all the memorable moments we lived through during my PhD time. They all greatly contributed to the successful completion of this dissertation.

Furthermore, I want to express my gratitude towards the Ecole Normale Supérieure de Cachan and the researchers from the Laboratoire Spécification et Vérification for making two incredible one-month research visits in Paris possible, and, moreover, to the German Academic Exchange Service (DAAD) within the Egide/DAAD Project *Procope 08/09* and the DFG Research Training Group *AlgoSyn* for their financial support during that unforgettable time.

My special thanks go to my officemate Stefan Rieger for all the exciting moments we had during our office time in Aachen and in several conference locations, and for always being a great, helpful, and cooperative friend. I am also grateful to all other colleagues from Lehrstuhl für Informatik 2 for the nice research environment, which contributed a lot to this dissertation.

I am especially indebted to Dr. Benedikt Bollig who reviewed major parts of this dissertation in all stages of maturity, and provided helpful feedback and many valuable suggestions for improvements. Great thanks also belong to all the other people—especially Professor Dr. Martin Leucker, Dr. Peter Habermehl, Cathrin Niebel, Daniel Retkowitz, David Piegdon, and Stefan Schulz—for revising substantial parts of this dissertation, discussing enhancements, and proposing useful changes.

Many thanks go to my student assistants David Piegdon and Stefan Schulz who greatly contributed to creating the learning library `libalf` and the learning application `Smyle` presented in this dissertation.

Last, but certainly not least, my special thanks go to my parents Dr. Wolfgang and Christa Kern, and my girlfriend Cathrin for always giving me so much love, support, and strength throughout all these fascinating and demanding years.

Carsten Kern

Contents

1 Introduction **1**
1.1 Formal Methods 1
1.2 Contribution 5
1.3 Outline 7

2 Preliminaries **9**
2.1 Basic Definitions 9
2.2 Words and Regular Languages 10
2.3 Finite-State Acceptors 11

I Inductive Learning Techniques and their Applications **19**

3 A Plethora of Learning Techniques **21**
3.1 The Big Picture 21
3.2 Offline Learning 24
3.3 Online Learning 33
3.4 Advantages and Disadvantages 42
3.5 Summary 43

4 Learning Nondeterministic Automata **45**
4.1 A Naïve Approach: Employing L* 46
4.2 NL*: From Tables to RFSA 47
4.3 NL*: The Algorithm 53
4.4 NL* vs. L*: an Example for an Exponential Gain 60
4.5 Other Approaches 62
4.6 UL*: Remodeling NL* for Learning Universal Automata 64
4.7 Experiments 66
4.8 Lessons Learned 70
4.9 Beyond Nondeterministic- and Universal Automata 73

5 Learning Congruence-Closed Languages **75**

6 Learning and Synthesis of Distributed Systems **81**
6.1 Preliminaries 82
6.2 Learning Communicating Finite-State Machines 94
6.3 Partial-Order Learning 104
6.4 Related Work 106
6.5 Summary 108

II Software Development Employing Learning Techniques 109

7 The `Smyle` Modeling Approach 111
7.1 Preliminaries 111
7.2 The `SMA` in Detail 116
7.3 `SMA` vs. Other Lifecycle Models 122
7.4 Exemplifying the `SMA` 126

III Tools and Applications 131

8 Learning Libraries 133
8.1 `libalf`: The Automata Learning Framework 134
8.2 Implementation Details 134
8.3 Learning Using the `libalf` Library 138

9 Synthesizing Models by Learning from Examples 143
9.1 `Smyle` from a User Perspective 143
9.2 Case Studies 152
9.3 Implementation Details 159
9.4 Further Areas of Application 164

10 Conclusions and Outlook 169

A Auxiliary Calculations for Chapter 2 171

B Insights on RFSA Learning (Interesting Examples) 173
B.1 An example where NL* needs more membership queries than L* 173
B.2 An example where NL* needs more equivalence queries than L* 177
B.3 An example where the intermediate hypothesis is not an RFSA 179
B.4 An example for non-termination of a row-based algorithm 181
B.5 An example for a non-increasing number of states 183
B.6 An example for a decreasing number of states 185
B.7 NL* might need more equivalence queries than minimal DFA has states . . 187

C An Algorithm for Solving the PDL Membership Problem 191

Bibliography 195

List of Figures 207

List of Tables 209

Index 211

1 Introduction

Nowadays, everyday life is almost totally dependent on software driven devices. Many of them are extremely safety-critical (also called life-critical) applications. Life-critical software is omnipresent. As such they can be found everywhere in daily life, e.g., within intensive care devices, safety devices in the automotive- or avionics sector, e.g., cruise controls, autonomous automobiles and autopilots, etc. Their malfunction could cause deep impact on our life, e.g., severe injuries or even death, but might also result in fatal damage to, or loss of, machinery, e.g., on extremely cost intensive space missions, or global environmental harm in case of a defective nuclear power plant, or misdirected nuclear or biochemical weapons. Therefore, strong attention must be payed to develop faultless software.

In the need of means to approach automated, error-free software development, usually two major approaches for tackling this problem are employed. Firstly, as code written by human beings is in many cases defective, (semi-) automatic generation of implementations seems preferable. Another technique to ensure software reliability is to check an abstract model of the system to be with respect to certain properties. To this end typically so-called *formal methods* are employed. Among the most important range: *model checking* [CGP99, BK08] and *theorem proving* [RV01]. In the following, we deliver some insights into the field of formal methods which allows for mathematical principles which in turn make mechanical treatment of software analysis possible.

1.1 Formal Methods

Formal methods are mathematical means for specifying, designing, developing and verifying software and hardware. Their major objective is to assure reliability and correctness of system implementations. As real world systems are usually by far too large to be checked for validity and correctness, abstract representations of these systems are needed. There are two main possibilities to obtain such an abstract model. Either it is constructed by hand which is an error-prone and tedious task, or mechanisms to (semi-) automatically generate a system abstraction or even code fragments are employed. If the automatic transformation from some kind of input (e.g., requirement specifications) is known or proven to be correct, the derived abstract model is so, too.

In this context the principle of *algorithmic synthesis* plays an important role as, on basis of formal methods, it allows for a mechanical transformation from some kind of formal specification, e.g., by a logic like LTL [Pnu77] or high-level specification languages like statecharts [Har87] or message sequence charts [ITU04] into a system implementation.

1.1.1 Algorithmic Synthesis

The problem of *algorithmic synthesis* ranges among the very important fields of computer science. It traces back to Alonzo Church and is known under the name *Church's problem* [Chu57], which states that "*Given a requirement … expressed in some suitable logistic* *system … find recursion equivalences representing … (it)*". In other words, synthesis is the process of transforming some kind of specification into an implementation being conform with it. This concept is highly desirable because a solution to it would feature an automated or semi-automated process to derive programs from specifications without having to manually develop them, and subsequently verify their correctness.

Although already more than half a century old, the field of algorithmic synthesis is a wide and active area of research. It ranges from *controller synthesis* [KPP09], i.e., the problem of restricting the internal behavior of a program to robustly conform to the specification no matter how the environment behaves, over approaches for *automata synthesis* [TB73] where from a given set of classified example words specifying a regular language, a finite-state automaton has to be synthesized that is consistent with the input, and *synthesizing open reactive systems* on basis of games [BSL04], to approaches for *real code synthesis*, e.g., from specification languages (like UML or Esterel) [WSK03, CDO97] or (timed) automata [BCG$^+$99, Amn03]. A recent, elaborate survey over the field of game-based synthesis of controllers and reactive systems is given by Thomas in [Tho09].

In this thesis we will address the synthesis of finite-state automata from sets of classified data, and, moreover, the synthesis of distributed systems in terms of communicating finite-state automata from scenarios which are given as message sequence charts. Let us, in the following, shortly introduce the formal models and methods of interest.

1.1.2 Modeling and Learning Finite-State Systems

Finite-state machines or *automata* are a common means for describing software (or hardware) systems on an abstract level of detail. There exists a plethora of extensions to finite-state automata, e.g., input/output-, timed-, or probabilistic variants. In their basic version, such automata consist of a finite set of locations (called *states*) the system may be in. At each point of time the system is located in exactly one such state. States are connected via action-labeled, directed edges (called *transitions*) which describe how the system evolves by changing from one state to another when reading an input action. Some states have additional properties of being *initial* or *final* (or both). Usually an automaton features only one initial state in which the modeled system starts its execution. Furthermore, being in a final state indicates that the system execution may terminate at this point. A sequence w of actions (i.e., a *word*) is *recognized* or *accepted* by an automaton, if there is a sequence of states (starting in an initial and ending in a final state) which are connected by transitions that are labeled by the actions of w succeeding in the correct order of occurrence in w.

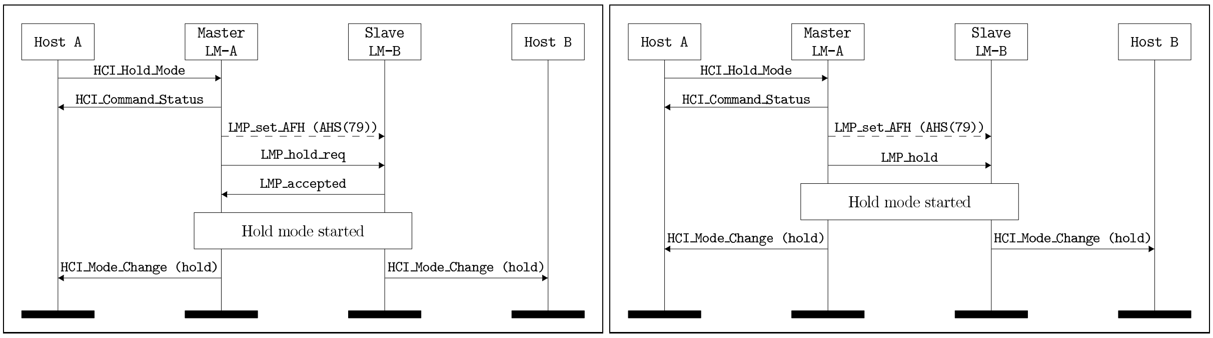
We usually distinguish between *deterministic* and *nondeterministic* versions of finite-state automata. In deterministic automata the next state is always uniquely determined by the current state and the next input symbol. In nondeterministic automata, however, given a current state and an input symbol, there is a choice of next states and therefore multiple possible continuations. Unlike deterministic automata, nondeterministic automata may have several initial states. Depending on the underlying application, one or the other model might be more suitable. Sometimes, for example, determinism of the automata is desirable, in other cases the size (i.e., the number of states of the automaton) might be tremendously relevant whereas determinism is of minor importance. As is well-known, nondeterministic finite-state automata can be exponentially more succinct than (minimal) deterministic automata that recognize the same set of words (also called a *language*) and, therefore, play an important role in several application areas where compact representations are desired.

As usually every software (or hardware) system can be regarded as a finite-state machine (because practically there are no infinite calculations possible), synthesis of such automata is of great interest because, as already mentioned, such abstractions of real world systems can be checked for correctness, or certain desired or undesired properties more easily. The field of automata synthesis, we referred to previously, will be one of the main focuses in this dissertation. More precisely, we will discuss so-called *learning algorithms* which take as input (i.e., as system specification) a set of classified words, i.e., words which are known to either be in the target language or not, and try to find a suitable finite-state automaton (i.e., an abstract system implementation) which behaves in accordance with these given words.

1.1.3 Modeling Communicating Systems

A second objective of this thesis is to investigate the learnability of communicating systems. In order to describe and analyze a distributed system's behavior formally, we need specification languages and models which are able to describe or express its essential communication properties. To this end, we will introduce *message sequence charts* as specification language and *communicating automata* as model for describing the output of distributed model synthesis.

Message Sequence Charts It is common design practice to start specifying a system's behavior in terms of simple pictures which describe the interaction of components. A widely accepted formal specification language for this purpose are *message sequence charts* (MSCs, for short). They are standardized by the International Communication Union ITU, and regularly revised and extended [ITU96, ITU98, ITU99, ITU04]. Moreover, they were adopted to the UML standard under the name *sequence diagrams* [Ara98, SP99] and are—thanks to their clear notation—famous among engineers, and, therefore, extensively used in industry. Visually, MSCs contain vertical lines denoting the communication entities (also called *processes*, *instances*, or *agents*) of the system on which time abstractly evolves from top to bottom, and, moreover, horizontal or slanted arrows expressing information flow or message exchange between processes. Formally, an MSC describes a *partial order*, i.e., a set of partially ordered events (i.e., sending or receiving events), describing which events happen before other events, and which ones are independent of others. E.g., all events along a process line are ordered such that events located higher on the process line are ordered before the lower ones, and the send event of a message is always ordered before its corresponding receive event. In Figure 1.1(a) an example of an MSC with four processes (`Host A`, `Host B`, `Master LM-A`, and `Slave LM-B`) is given. E.g., the sending event of message `HCI_Hold_Mode` is ordered before the receive event of message `HCI_Command_Status`, and both are ordered before the receiving event `HCI_Mode_Change (hold)` on process `Host A`. In contrast the receiving events `HCI_Command_Status` on process `Host A` and `HCI_Mode_Change (hold)` on process `Host B` are unordered according to the MSC semantics. Note that MSCs abstract from internal behavior of processes and only concentrate on the message exchange between processes. As such, MSCs are to be seen as partial specifications (also called *scenarios*) of real system runs. Due to their appealing structure, they are employed in many phases of the software development process, e.g., for requirements specification, system validation and verification, and conformance testing, etc.



(a) Host requesting `hold` mode (b) Host forcing `hold` mode

Figure 1.1: Two MSCs from the Bluetooth® (version 3.0) specification

The ITU Z.120 standard [ITU04] features a textual grammar description of MSCs and a graphical grammar. Moreover, a clear formal denotational semantics is described in [Ren98]. These standardized works are premises for easing mechanical treatment of specifications, making MSCs interesting for tool development based on formal methods. Indeed, there exists a variety of applications [AHP96, BAL98, BKSS06] for analyzing MSCs or sets of MSCs for detecting implementation deficiencies like *non-local choice* [BAL97] and *race conditions* [AHP96, EGP07], or for assuring their *implementability* [AEY03, Loh03, AEY05].

To convince ourselves of the real-world applicability of MSCs, let us now consider an industrial example for the use of MSCs within the current Bluetooth® specification [Blu09] from April 2009. Bluetooth® is a standard for wireless communication initially introduced by five companies (Ericsson, Nokia, IBM, Toshiba, Intel) forming the Bluetooth® Special Interest Group (SIG). The Bluetooth® specification manual [Blu09] features a whole range of requirements recorded as MSCs. In Figure 1.1 two such MSCs are depicted. They describe different possibilities—either after a negotiation phase or in a forced fashion—how a host (`Host A`) can place a device into *hold mode* using the `HCI_Hold_Mode` command. The purpose of this hold mode is to either make free capacity available, or to perform actions like net scanning, paging, or attending another so-called *piconet*, which is a collection of a bounded number of devices connected via Bluetooth® in an ad hoc fashion.

Being in state *connected*, the ACL logical transport (which carries control information interchanged between the link managers of master and slave device(s)) to a slave can be put to hold mode for a specific time via the message `LMP_hold_req` (cf. Figure 1.1(a)) or `LMP_hold` (cf. Figure 1.1(b))—both with parameter *hold time* or *hold instant* which are omitted in the MSC representation. Therefore, initially the host control interface (`HCI`) sends a `HCI_Hold_Mode` message telling the master that it wants to enter the hold mode. The master returns an acknowledgment message (`HCI_Command_Status`) and, moreover, informs the slave (via messages `LMP_set_AFH` and `LMP_hold_req` (cf. Figure 1.1(a)), or messages `LMP_set_AFH` and `LMP_hold` in case of forced hold (cf. Figure 1.1(b))) that the hold mode should be entered. While in the second operation mode the slave is forced to go into hold mode using message `LMP_hold`, in the first case master and slave negotiate about entering this mode and the final answer is returned to the master (cf. messages `LMP_hold_req` and `LMP_accepted`). The hold mode now has started. Thus, master and slave, respectively notify their hosts via the `HCI_Mode_Change` message that the hold mode was successively entered.

Communicating systems If we want to describe communicating systems formally, we have to choose a model which features local components, i.e., processes, that can interact or communicate with one another via some communication medium. For our purposes in this thesis, the model of *communicating finite-state machines* (CFMs) seems to be sensible, as processes are described by finite-state automata, a well-known and well-understood formalism, and these components can communicate via a priori unbounded order-preserving channels. This way, we are able to model asynchronous communication behavior.

The formal semantics of CFMs can be defined in terms of MSCs. Every local finite-state automaton acts as one process sending and receiving messages to and from other processes. In this way, the finite-state automata exchange messages over the CFM buffers. A CFM may accept when all local machines are located in final states and all buffers are empty. This implicitly ensures, that CFMs only accept MSCs and not some partial executions thereof.

As the model of CFMs is very expressive, making certain considerations intractable, we will restrict this model in a later chapter to achieve decidability.

A great advantage of communicating finite-state machines is that, though still an abstract formal model, they are closely related to real implementations as they are already distributed, and the local components are rather straightforward to realize. Thus, when we have explained how to synthesize these automata, we will also discuss in which way they are of avail within the software development cycle.

1.2 Contribution

The results of this dissertation are two-fold. On the one hand, inductive learning techniques are extended and two new inference algorithms for inferring nondeterministic, and universal, respectively, finite-state automata are presented. On the other hand, certain learning techniques are employed and enhanced to semi-automatically infer communicating automata (also called *design models* in the software development cycle). For both topics, theoretical results on the feasibility of the approaches, as well as an implementation are presented which, in both cases, support our theory.

Concerning the first objective to derive a so-called *active online learning algorithm* for nondeterministic finite-state automata (NFA), we present, in analogy to Angluin's famous learning algorithm L* [Ang87a] for deterministic finite-state automata (DFA), a version for inferring a certain subclass of NFA. The automata from this class are called *residual finite-state automata* (RFSA). It was shown by Denis et al. [DLT02] that there is an exponential gap between the size of minimal DFA and their corresponding minimal RFSA. Even if there are also cases where the canonical (i.e., minimal) RFSA is exponentially larger than a corresponding minimal NFA, we show that the new learning algorithm—called NL*—is a great improvement compared to L* as the inferred canonical RFSA has always at most the size of the corresponding minimal DFA but is usually even considerably smaller and more easy to learn. Unlike a learning procedure developed by Denis et al.—called DeLeTe2 [DLT04]—our algorithm is capable of deriving canonical RFSA. Like L*, the new algorithm will be applicable in many fields including pattern recognition, computational linguistics and biology, speech recognition, and verification. From our point of view, NL* might especially play a mayor role in the area of verification where the size of the models that are processed is of enormous importance and nondeterminism not regarded an unpleasant property.

The second objective of this thesis is to create a method for inferring distributed design models (CFMs) from a given set of requirements specified as scenarios (message sequence charts). The main idea is to extend the L* algorithm to cope with valid and invalid sets of system runs and, after some iterations, come up with an intermediate design model (a DFA) which exhibits features that make it distributable into communicating components (or processes) interacting via FIFO channels. Theoretical results on which classes of CFMs are learnable and corresponding time-complexity bounds are presented. We also developed a tool implementation called `Smyle`, realizing important theoretical results evolving from this part of the thesis, and applied it to several examples. Based on this learning formalism we also derive a software-engineering lifecycle model called the `Smyle` *Modeling Approach* in which we embedded our learning approach.

Additionally, we launched a project for a new learning library called `libalf` which includes most of the learning algorithms (and their extensions) mentioned in this thesis.

Parts of this thesis were published in several technical reports, conference proceedings, and a journal [BKSS06, BKKL06, BKKL07, BKKL08b, BKKL08a, BHKL08, BKKL09, BHKL09, BKKL10]. An article on learning communicating automata from MSCs has been accepted at IEEE Transactions on Software Engineering (TSE) [BKKL]. A detailed overview is given in the following list of publications.

[BKSS06]	Benedikt Bollig, Carsten Kern, Markus Schlütter, and Volker Stolz. “MSCan: A tool for analyzing MSC specifications”. (*TACAS 2006*). This paper describes the tool `MSCan` for analyzing MSC specifications with regard to potential implementation deficiencies of distributed systems. `MSCan` is embedded into the software engineering lifecycle model introduced in Chapter 7.
[BKKL07]	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. “Replaying play in and play out: Synthesis of design models from scenarios by learning”. (*TACAS 2007*). A completely revised version including several extensions is included in this thesis as Chapters 6 and 9, and will be published as [BKKL].
[BKKL08a]	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. “*Smyle*: A Tool for Synthesizing Distributed Models from Scenarios by Learning”. (*CONCUR 2008*). The paper describes the learning application `Smyle` which is capable of inferring CFMs from scenarios specified as MSCs. Some parts of this paper were integrated into Chapter 9 of this work.
[BKKL09]	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. “SMA—The Smyle Modeling Approach”. (*CEE-SET 2008* (IFIP), to appear). It describes the `SMA` lifecycle model embedding the approach from [BKKL07] into a software development lifecycle.
[BHKL09]	Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. “Angluin-Style Learning of NFA”. (*IJCAI 2009*, to appear). This paper introduces a new inference algorithm for NFA along the lines of Angluin’s L* algorithm for learning DFA. A substantial extension featuring a learning algorithm for universal automata and more sophisticated statistical results are included in Chapter 4 of this thesis.

[BKKL10]	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. "SMA—The Smyle Modeling Approach". (*Computing and Informatics* Journal, 2010, to appear). This article is an extension of [BKKL08b] and [BKKL09], and is based on Chapter 7 of this thesis.
[BKKL]	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering*. To appear.

1.3 Outline

The rest of this thesis is organized as follows: the next chapter introduces basic notions and notations for regular languages and their acceptors which will be needed in subsequent chapters. After this preliminary chapter, the thesis is divided into three main parts. Part I describes inductive learning techniques and applications thereof, Part II shows how to embed learning into software development yielding a new software development lifecycle model, and the final part, Part III, introduces tools that implement our theoretical results.

To be more precise, Chapters 3 to 4 are concerned with fundamental learning algorithms. These algorithms get certain classified input and aim at deriving a model (i.e., a finite-state automaton) being conform with the input. To this end, in Chapter 3 we introduce several inference algorithms which are regularly employed in all kinds of applications. As for our purposes one of these algorithms—namely, Angluin's active online learning algorithm L^*—is of special interest for later practical and theoretical results, we give a more detailed report on L^*. In analogy to L^*, we will develop a new active online learning algorithm, called NL^*, for inferring nondeterministic automata in Chapter 4. It is important to mention that this algorithm is the first of its kind. Moreover, we will see that it is efficient, and, even though theoretical complexity results are slightly worse, in the experiments described in Section 4.7 NL^* outperforms L^* by far. In Chapter 5 we will present an optimization called *congruence-closed language learning* that may substantially reduce memory consumption of learning algorithms like L^* or NL^*. Chapter 6 puts learning into the context of distributed system synthesis. There, we identify classes of regular languages that are learnable using an extension of the L^* algorithm which, afterwards, will be improved applying the concept from Chapter 5.

Part II—represented by Chapter 7—starts with the introduction of a logic which subsequently will be of help to reduce the number of questions a potential user will be asked during a learning phase when using our inference approach. Afterwards, we introduce a new software development lifecycle model, called the `Smyle` *Modeling Approach* (`SMA`, for short). In this lifecycle model, we exploit the learning techniques introduced in the chapters before to obtain a system implementation based on an interactive process in which scenarios are presented to, and classified by, the user. Moreover, we show how to ease this task by employing the logic introduced in the preliminaries of this chapter. Subsequently, we compare the `SMA` to well-known lifecycle models like the waterfall model, the V-model, or rapid prototyping, followed by a small example demonstrating how the learning-based part of `SMA` works in practice.

Finally, Part III is dedicated to the tools `libalf` and `Smyle` developed as part of this thesis. To this end, Chapter 8 introduces the learning library `libalf` which currently implements most learning techniques presented in this thesis. It is meant to be an ongoing project of several universities (currently: RWTH Aachen University, École Normale

Supérieure de Cachan, and TU Munich) in which we provide classical and new learning algorithms in an open source project to the public. We hope that `libalf` will find a broad acceptance among researchers, and that it will be augmented with new algorithms from different universities in near future. Thereafter, Chapter 9 presents the implementation of the results obtained in Chapter 6. The tool `Smyle` synthesizes distributed automata from given scenarios by employing an extension of Angluin's learning approach. Having described the tool in detail, we give some small- to mid-size case studies, followed by details concerning the implementation, and close this chapter by proposing some interesting ideas for further applications on basis of `Smyle`.

Chapter 10 concludes this thesis by summarizing its main contributions and giving some ideas and prospects for future work.

The appendix features, among other things, several examples that give insights into the new inference algorithm NL^*. Inter alia, it comprises examples where NL^* behaves "worse" than L^*, cases in which the number of states remains equal or even decreases (which cannot happen in Angluin's L^* algorithm), and exemplarily shows that a naïve implementation of NL^* leads to a non-terminating algorithm.

2 Preliminaries

In this chapter we present some basic terminology, definitions, and concepts of formal language theory as basis for subsequent chapters. For a complete overview over the basics of formal language theory we refer to [HMU06]. For a nice overview over the field of complexity theory, which may be of use at some points in this thesis, the interested reader might consult [Sip05].

2.1 Basic Definitions

As usual, let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers. If S is a set, we denote $|S| \in \mathbb{N} \cup \{\infty\}$ the size of S, i.e., the number of elements contained in S.

Definition 2.1.1 (Power set). *The* power set 2^S *of a set* S *is the set of all subsets of* S. *More formally:* $2^S = \{P \mid P \subseteq S\}$.

Definition 2.1.2 (Relation). *Let* $n \in \mathbb{N}$ *and* S *be a set. An* n*-ary* relation R *over* S *is a set* $R \subseteq \underbrace{S \times \ldots \times S}_{n}$. R *is called* binary *if* $n = 2$.

Definition 2.1.3 (Equivalence relation). *Let* S *be a set. A binary relation* $\sim$ *over* S *is called an* equivalence relation *over* S *if it fulfills the following properties for all elements* $a, b, c \in S$:

- $(a, a) \in \sim$ *(Reflexivity)*
- $(a, b) \in \sim \Rightarrow (b, a) \in \sim$ *(Symmetry)*
- $(a, b) \in \sim \wedge (b, c) \in \sim \Rightarrow (a, c) \in \sim$ *(Transitivity)*

For the sake of readability, we sometimes prefer to use the infix notation $a \sim b$ *to denote* $(a, b) \in \sim$.

On basis of an equivalence relation $\sim$ over a set S, we can partition S into so-called *equivalence classes* for which each such class contains all equivalent elements of one kind.

Definition 2.1.4 (Equivalence class). *Let* S *be a set and* $\sim$ *be an equivalence relation over* S. *The* equivalence class *of* $a \in S$ *wrt.* $\sim$ *(written* $[a]_\sim$*) contains all elements of* S *equivalent to* a, *i.e.,* $[a]_\sim = \{b \in S \mid a \sim b\}$. *The set of all such equivalence classes is denoted by* $S_{/\sim} = \{[a]_\sim \mid a \in S\}$.

If an equivalence relation respects certain operations, it is called a congruence wrt. these operations.

Definition 2.1.5 (Congruence relation)**.** *Let S be a set and $f : S \times \cdots \times S \to S$ be an n-ary function over S. An equivalence relation $\sim$ over S is called a* congruence relation *over S wrt. f if the following implication holds:*

$$\forall a_1, \ldots, a_n, b_1, \ldots, b_n \in S : \; [(a_1, b_1), \ldots, (a_n, b_n) \in \sim \Rightarrow f(a_1, \ldots, a_n) \sim f(b_1, \ldots, b_n)] .$$

If f is binary and $\forall a, b, c \in S$: $a \sim b \Rightarrow f(a, c) \sim f(b, c)$, we call $\sim$ a right congruence relation *over S wrt. f.*

2.2 Words and Regular Languages

In this subsection we define the notion of *words* and *regular languages.*

Let Σ be a finite, non-empty set, also called an *alphabet*. Elements of Σ are called letters. A *word* is a linear sequence of letters and a set of such words is called a *word language* or language, for short. We distinguish the empty letter sequence denoted by ε and call it the *empty word.*

The length $n \in \mathbb{N}$ of a word $w = a_1 \ldots a_n \in \Sigma^*$ ($a_i \in \Sigma, i \in \{1, \ldots, n\}$) is denoted as $|w|$ and specifies the number of letters w consists of. Thus, the empty word, for example, has length zero, i.e., $|\varepsilon| = 0$. The set of words of length n is denoted $\Sigma^n := \{w \in \Sigma^* \mid |w| = n\}$ and the set of all words over alphabet Σ is abbreviated $\Sigma^* := \bigcup_{i \in \mathbb{N}} \Sigma^i$. Moreover, $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of all words of positive length.

Having defined words over Σ as sequences of letters allows us to access the i-th letter ($i \in \{1, \ldots, n\}$) of a word w of length n via $w[i]$. Usually, we range over letters by $a, b, c, \ldots$, over words by $u, v, w, \ldots$ and over (word) languages by L, L', etc.

Definition 2.2.1 (Concatenation)**.** *We set $u \cdot v$ to be the word w of length $|u| + |v|$. The* concatenation *$\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ is the binary operation defined by $u \cdot v := w$ such that for words u, v of length n and m, respectively, $w[i] = u[i]$ for $1 \leq i \leq n$ and $w[i] = v[i]$ for $n + 1 \leq i \leq n + m$. For $n \in \mathbb{N}$ and $L \subseteq \Sigma^*$, we define L^n to be the set $\{w \in \Sigma^* \mid w = w_1 \cdot \ldots \cdot w_n \, , \; w_i \in L \text{ for } 1 \leq i \leq n\}$. Note that instead of $u \cdot v$ we usually only write uv.*

On basis of the concatenation function, we can define the *prefix-* and *suffix-set* of a given word $w \in \Sigma^*$: *pref*$(w) = \{u \in \Sigma^* \mid w = uv \text{ for some } v \in \Sigma^*\}$ and *suff*$(w) = \{v \in \Sigma^* \mid w = uv \text{ for some } u \in \Sigma^*\}$, respectively. This definition can be lifted to sets of words $S \subseteq \Sigma^*$: *pref*$(S) = \bigcup_{w \in S}$ *pref*(w) and *suff*$(S) = \bigcup_{w \in S}$ *suff*(w).

Definition 2.2.2 (Length-lexicographical order)**.** *Given a total order $<_{\text{lex}}$ over Σ, we extend it to words and obtain a* canonical total order $<_{\text{lex}}$ *over Σ^*. For $u, v \in \Sigma^*$, $u <_{\text{lex}} v$ iff $|u| < |v|$ or ($|u| = |v| = n$ and there exists $k \in \{1, \ldots, n\}$ such that, for all $1 \leq i < k$, $u[i] = v[i]$ and $u[k] <_{\text{lex}} v[k]$). We call this order the* length-lexicographical order *(or lexicographical order, for short) over Σ^*.*

We now define a congruence relation which will be important in the next chapters because it helps to classify languages as *regular languages* and is the basis of many learning algorithms presented in Chapters 3 and 4.

Definition 2.2.3 (*Nerode right congruence*)**.** *Let $L \subseteq \Sigma^*$ be a language. Then we define the following binary relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$:*

$$\text{For all words } u, v \in \Sigma^* : u \sim_L v \text{ iff } \forall \, w \in \Sigma^* : uw \in L \Longleftrightarrow vw \in L.$$

The relation $\sim_L$ *is a right congruence relation over* Σ^* *wrt. concatenation and called* Nerode right congruence. *In case* $(u, v) \in \sim_L$*, we call* u *and* v L*-equivalent.*

We denote the number of equivalence classes of a language L by $\mathit{index}(L) = |\Sigma^*_{/\sim_L}| = |\{[x]_{\sim_L} \mid x \in L\}|$, called the *index of* L. L is said to be of *finite index* if $\mathit{index}(L)$ is finite. In general, for arbitrary languages this index can be infinite. We now define the class of *regular languages* to be the set of languages which are of finite index.

Definition 2.2.4. $L \subseteq \Sigma^*$ *is a* regular language *iff* $\sim_L$ *has finite index.*

We introduce *regular expressions* as an easy means for describing regular languages.

Definition 2.2.5 (Regular expressions)**.** *The set of* regular expressions $\mathfrak{R}$ *over* Σ *is inductively defined as follows:*

- $\Lambda \in \mathfrak{R}$*,*
- $a \in \mathfrak{R}$ *for* $a \in \Sigma$*,*
- *if* $r, r' \in \mathfrak{R}$ *then* $r \cdot r'$*,* $r|r'$ *and* r^* *are elements of* $\mathfrak{R}$*.*

The semantics $[\![.]\!] : \mathfrak{R} \to 2^{\Sigma^*}$ *of regular expressions over* Σ *is defined as:*

- $[\![\Lambda]\!] = \emptyset$*,*
- $[\![a]\!] = \{a\}$ *for* $a \in \Sigma$*,*
- *if* $r, r' \in \mathfrak{R}$ *then:*
    - $[\![r \cdot r']\!] = [\![r]\!] \cdot [\![r']\!] = \{w \cdot w' \mid w \in [\![r]\!] \text{ and } w' \in [\![r']\!]\}$*,*
    - $[\![r|r']\!] = [\![r]\!] \cup [\![r']\!]$*, and*
    - $[\![r^*]\!] = \bigcup_{i \in \mathbb{N}} [\![r]\!]^i$

  are elements of $\mathfrak{R}$*. Note that* $[\![\Lambda^*]\!] = \{\varepsilon\}$*.*

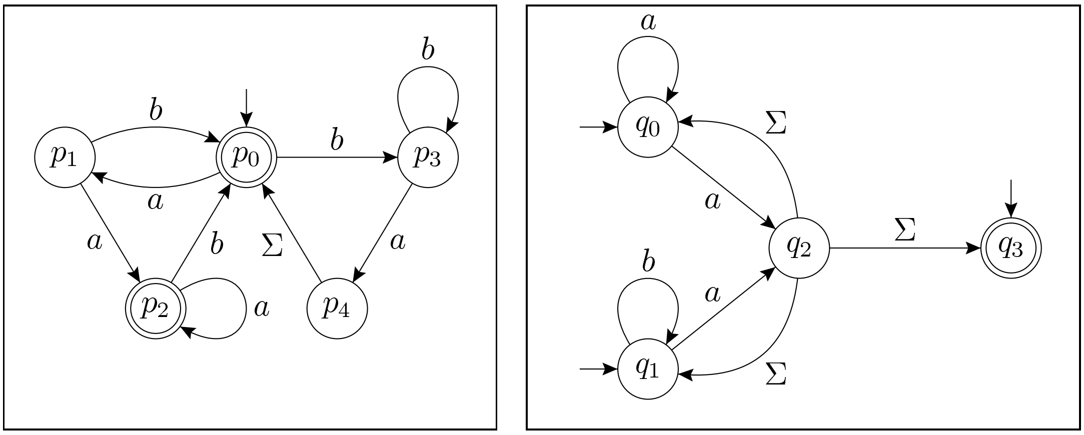
We define the language of a regular expression $r \in \mathfrak{R}$ to be $L(r) = [\![r]\!]$. As we will see later in this chapter, the class of regular expressions defines the class of regular languages. Let us introduce the notion of *residual languages* of a given language L which plays an important role in the context of learning algorithms, as residual languages allow for a elegant characterization of regular languages. More precisely, to infer a target language L, learning algorithms try to identify words characterizing equal residual languages of L.

Definition 2.2.6 (Residual language)**.** *For a language* $L \subseteq \Sigma^*$ *and* $u \in \Sigma^*$*, we denote by* $u^{-1}L$ *the set* $\{v \in \Sigma^* \mid uv \in L\}$*.* $L' \subseteq \Sigma^*$ *is a* residual language *of* L *if there is* $u \in \Sigma^*$ *with* $L' = u^{-1}L$*. We denote by* $\mathit{Res}(L)$ *the set of residual languages of* L*.*

For simplicity, we also talk of a *residual* rather than of a residual language. Note that $u \sim_L v$ iff $u^{-1}L = v^{-1}L$.

2.3 Finite-State Acceptors

In this paragraph, we will introduce several classes of finite-state automata as acceptors for regular languages.



(a) Minimal DFA $\mathcal{A}$ for L (b) NFA $\mathcal{A}'$ for L with less states than $\mathcal{A}$

Figure 2.1: Finite-state acceptors for regular language $L = L(((a^*|b^*)a\Sigma)^*)$ $(\Sigma = \{a, b\})$

2.3.1 Deterministic and Nondeterministic Finite-State Automata

A prominent concept for representing regular languages are finite-state automata. Typically, one distinguishes *deterministic* and *nondeterministic* versions thereof.

Definition 2.3.1 (Finite-state automaton)**.** *A* (nondeterministic) finite-state automaton *(NFA) is a tuple* $\mathcal{A} = (Q, Q_0, \delta, F)$ *over* Σ *where:*

- Q *is a finite set of* states*,*
- Q_0 *is a set of* initial states $Q_0 \subseteq Q$*,*
- $\delta : Q \times \Sigma \to 2^Q$ *is the* transition function*,*
- $F \subseteq Q$ *is a set of* final states*.*

We call $\mathcal{A}$ a *deterministic finite-state automaton* (DFA) if $|Q_0| = 1$ and $|\delta(q, a)| = 1$ for all $q \in Q$ and $a \in \Sigma$. The transition function δ of an NFA is extended as usual to $\bar{\delta} : Q \times \Sigma^* \to 2^Q$ by $\bar{\delta}(q, \varepsilon) = \{q\}$ and $\bar{\delta}(q, aw) = \bigcup_{q' \in \delta(q,a)} \bar{\delta}(q', w)$, and subsequently to $\hat{\delta} : 2^Q \times \Sigma^* \to 2^Q$, i.e., to sets of states $Q' \subseteq Q$, by $\hat{\delta}(Q', w) = \bigcup_{q \in Q'} \bar{\delta}(q, w)$. To simplify notation, we use δ to denote both $\bar{\delta}$ and $\hat{\delta}$.

The *size* of a finite-state automaton $\mathcal{A} = (Q, Q_0, \delta, F)$ is defined as the number of its states: $|\mathcal{A}| := |Q|$. The *language of a state* $q \in Q$, denoted by L_q, is the set of words $w \in \Sigma^*$ such that $\delta(q, w) \cap F \neq \emptyset$. The *language* $L(\mathcal{A})$ *accepted* by $\mathcal{A}$ is the union of languages of its initial states: $L(\mathcal{A}) = \bigcup_{q \in Q_0} L_q$. We say that a word $w = a_1 \dots a_n \in \Sigma^*$ ($a_i \in \Sigma$, for $1 \leq i \leq n$) is accepted by $\mathcal{A}$ if it is contained in its language $L(\mathcal{A})$, or equivalently, if there is a sequence $q_0 a_1 q_1 a_2 \dots q_{n-1} a_n q_n$, where $q_0 \in Q_0$, $q_n \in F$, and $q_i \in \delta(q_{i-1}, a_i)$, for $1 \leq i \leq n$. Two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are called *equivalent* if they accept the same language, i.e., if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

We are now introducing the notion of *minimality* of an automaton with respect to some class of acceptors $\mathcal{C}$.

Definition 2.3.2 (Minimal automaton)**.** *Let* $\mathcal{C}$ *be a class of NFA over* Σ *and* $\mathcal{A} = (Q, Q_0, \delta, F) \in \mathcal{C}$*.* $\mathcal{A}$ *is called* minimal *for* $\mathcal{C}$ *if there is no equivalent automaton* $\mathcal{A}' = (Q', Q'_0, \delta', F') \in \mathcal{C}$ *with strictly fewer states than* $\mathcal{A}$*, i.e., with* $|Q'| < |Q|$*.*

E.g., we call a DFA or NFA, respectively, *minimal* if there is no equivalent DFA or NFA, respectively, with strictly fewer states. If the class we are referring to is clear from the context, we usually omit it. Note, in contrast to NFA, a DFA has always a unique minimal representative in its class:

Theorem 2.3.3 (Myhill-Nerode)**.** *For each regular language L, there exists a unique (up to isomorphism) minimal DFA $\mathcal{A}$ with $L(\mathcal{A}) = L$.*

Example 2.3.4. Let us consider the finite-state automata $\mathcal{A} = (Q, Q_0, \delta, F)$ and $\mathcal{A}' = (Q', Q_0', \delta', F')$ over $\Sigma = \{a, b\}$ from Figure 2.1 accepting the regular language $L = L(((a^*|b^*)a\Sigma)^*)$. I.e., language L recognizes all words that are multiples ($\in \mathbb{N}$) of: an arbitrary number of a's or b's followed by an a followed by an a or b. As $|Q_0| = \{p_0\} = 1$ and for every state $p \in Q$ and every letter $l \in \Sigma$ it holds $|\delta(q, l)| = 1$, automaton $\mathcal{A}$ is deterministic. Moreover, using techniques which are for example described in [HMU06], the reader may verify that $\mathcal{A}$ is a minimal DFA of its class. At the same time, automaton $\mathcal{A}'$ recognizes L, too. This automaton is obviously not deterministic, as it has several initial states (in fact: $Q_0' = \{q_0, q_1, q_3\}$) and, e.g., $|\delta'(q_0, a)| = 2$. Moreover, we see that an NFA can be smaller than the equivalent minimal DFA. In fact, an NFA for a regular language L can be exponentially more succinct than the equivalent minimal DFA accepting L. An example of such a language will be given in Chapter 4. ◇

Let us summarize some important theorems about the connection between regular languages and classes of acceptors that recognize regular languages. The following results are due to [Kle56] and [Ner58].

Theorem 2.3.5. *Let $L \subseteq \Sigma^*$ be a language, then the following statements are equivalent:*

(i) L is a regular language,

(ii) there exists a regular expression $r \in \mathfrak{R}$ with $L = L(r)$,

(iii) there exists a DFA $\mathcal{A}$ with $L = L(\mathcal{A})$, and

(iv) there exists an NFA $\mathcal{A}'$ with $L = L(\mathcal{A}')$.

In the following, we define the minimal DFA $\mathcal{A}_L$ of a regular language L. To this end, we make use of Nerode's right congruence $\sim_L$.

Definition 2.3.6 (The automaton $\mathcal{A}_L$)**.** *Based on Nerode's right congruence, for every regular language L we can derive a unique (or canonical) DFA $\mathcal{A}_L = (Q_L, \{q_0^L\}, \delta_L, F_L)$ with a minimal number of states, where:*

- $Q_L = \{[x]_{\sim_L} \mid x \in \Sigma^*\}$ *is the set of states,*
- $q_0^L = [\varepsilon]_{\sim_L}$ *is the initial state,*
- $\delta_L([x]_{\sim_L}, a) = \{[xa]_{\sim_L}\}$ *describes the transition function, and*
- $F_L = \{[x]_{\sim_L} \in Q_L \mid x \in L\}$ *is the set of final states.*

Moreover, on basis of residual languages of a regular language L defined in the previous section, we can also define the canonical DFA $\mathcal{A}(L)$ recognizing L. As we will subsequently see, the DFA $\mathcal{A}_L$ and $\mathcal{A}(L)$ are isomorphic and thus recognize the same language.

Definition 2.3.7 (The canonical DFA $\mathcal{A}(L)$)**.** *Let L be a regular language. The canonical DFA of L, denoted by $\mathcal{A}(L)$, is the tuple (Q, Q_0, δ, F) where:*

- *$Q = Res(L)$, i.e., the residuals exactly correspond to the states of $\mathcal{A}$,*
- *$Q_0 = \{L\}$,*
- *$\delta(L', a) = \{a^{-1}L'\}$ for $L' \in Q$ and $a \in \Sigma$,*
- *$F = \{L' \in Q \mid \varepsilon \in L'\}$.*

Let us now establish the connection between the automata $\mathcal{A}_L$ and $\mathcal{A}(L)$.

Theorem 2.3.8. *Let L be a regular language, then:*

(i) $L = L(\mathcal{A}_L) = L(\mathcal{A}(L))$,

(ii) the automata $\mathcal{A}_L$ and $\mathcal{A}(L)$ are isomorphic, and

(iii) the automaton $\mathcal{A}_L$ is minimal for the class of DFA.

Example 2.3.9. Let us derive the residual languages of regular language $L = L(r)$ recognized by automaton $\mathcal{A}$ from Figure 2.1(a). Performing some simple calculations yields: $\varepsilon^{-1}L = L = L_{p_0}$, $a^{-1}L = L_{p_1}$, $b^{-1}L = L_{p_3}$, $(aa)^{-1}L = L_{p_2}$, $(ab)^{-1}L = L_{p_0}$, and $(ba)^{-1}L = L_{p_4}$.

We see, that every state of $\mathcal{A}$ corresponds to a residual language of L. A short derivation of the above equations can be found in Appendix A.0.1. ◇

As it will play an important role in some learning algorithms presented in Chapter 3, we introduce the *prefix automaton* of a given finite set S.

Definition 2.3.10 (Prefix automaton)**.** *Given a finite set S of words over an alphabet Σ, the* prefix automaton *(also called* prefix tree acceptor*) is the (tree-like) DFA $\mathcal{A}(S) = (Q_+, \{q_0^+\}, \delta_+, F_+)$ whose states correspond to the prefixes of S and that exactly recognizes all words from S:*

- *$Q_+ = pref(S) \uplus \{q_{sink}\}$,*
- *$q_0^+ = \varepsilon$,*
- *$\delta_+(u, a) = \begin{cases} \{ua\} & , \text{ if } ua \in pref(S) \\ \{q_{sink}\} & , \text{ otherwise} \end{cases}$, and*
- *$F_+ = S$.*

For the sake of clarity, we usually omit the sink state q_{sink} as in Figure 2.2.

Example 2.3.11. Let $S = \{aaa, ab, bb\}$. The prefix tree acceptor for S is presented in Figure 2.2. Formally, it consists of states $Q_+ = \{\varepsilon, a, aa, aaa, ab, b, bb, q_{\text{sink}}\}$, the initial state $q_0^+ = \varepsilon$, transition function δ as defined above augmented by transitions to the sink state q_{sink}, and the set of final states $F_+ = \{aaa, ab, bb\}$. As the prefix automaton is a deterministic finite-state automaton recognizing a finite regular language, the residuals are easy to determine. The residual for state ε, for example, is $\varepsilon^{-1}L = \{aaa, ab, bb\} = L$, the residual for state a is $a^{-1}L = \{aa, b\}$ and the residual of state bb is $(bb)^{-1}L = \{\varepsilon\}$. ◇

As we will see in the following, there exists a class of automata called RFSA which are a subclass of NFA but subsume the class of DFA and, like DFA, have a unique minimal representative for each regular language.

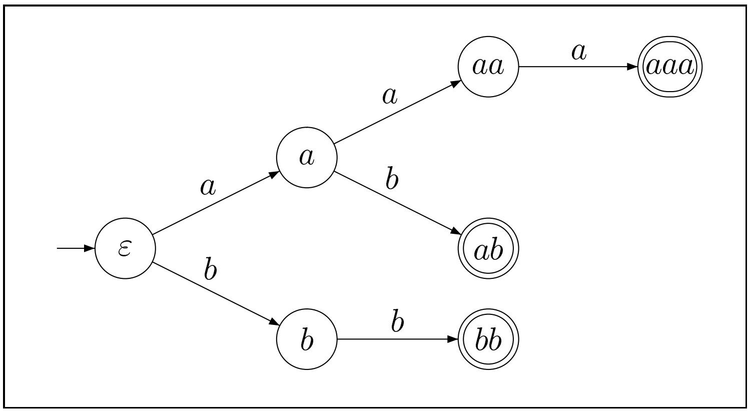
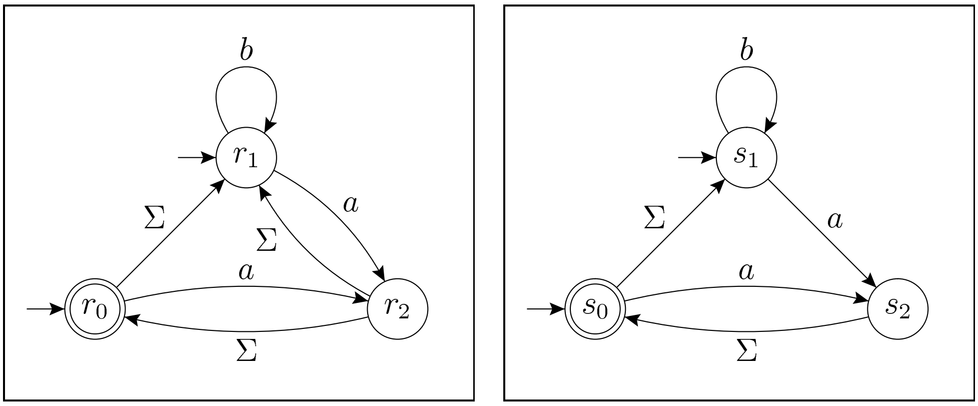


Figure 2.2: Prefix tree acceptor for $S = \{aaa, ab, bb\}$ without sink state q_{sink}

2.3.2 Residual Finite-State Automata

Here we recall the notion of *residual finite-state automata* (RFSA) introduced and studied in the work of Denis et al. [DLT02]. RFSA are a subclass of NFA inheriting some desirable features of DFA. Most important for the purpose of learning, which we will consider in detail throughout the next chapters, every regular language is accepted by a unique (canonical) RFSA with a minimal number of states (i.e., it is minimal in the class of RFSA). As this property does not hold for arbitrary NFA, it seems difficult to come up with learning algorithms for the whole class of NFA. At the same time, like for NFA, RFSA can be exponentially more succinct than DFA, making it the preferable automaton model to work with in practical learning applications.

Technically, RFSA and DFA have the property that the states of the automata correspond to residual languages (cf. Definition 2.2.6). This is not true for all NFA.



(a) Canonical RFSA $\mathcal{R}_L$ (b) NFA $\mathcal{R}'_L$ with less transitions than $\mathcal{R}_L$

Figure 2.3: Nondeterministic acceptors for regular language L from Figure 2.1

Example 2.3.12. Consider the NFA $\mathcal{R}'_L$ in Figure 2.3(b). Let us first determine the residuals of this automaton. Applying results from [Ard60, Ard61] concerning solutions to regular equations, we get: $L_{s_0} = \varepsilon^{-1}L = L, L_{s_1} = b^{-1}L$, and $L_{s_2} = (ba)^{-1}L$. See Appendix A.0.2 for the derivation of the residuals. Similar calculations yield the same results for the languages of the states of automaton $\mathcal{R}_L$ from Figure 2.3(a). ◇

Note that, with Theorems 2.3.3 and 2.3.8, for a minimal DFA $\mathcal{A}$, there is a natural bijection between its states and the residual languages of $L(\mathcal{A})$.

Definition 2.3.13 (Residual finite-state automaton)**.** *A residual finite-state automaton (RFSA) over Σ is an NFA $\mathcal{R} = (Q, Q_0, \delta, F)$ over Σ such that for each $q \in Q$, $L_q \in \mathit{Res}(L(\mathcal{R}))$.*

In other words, each state accepts a residual language of $L(\mathcal{R})$, but not every residual language must be accepted by a *single* state. Intuitively, the states of an RFSA are a subset of the states of the corresponding minimal DFA. Yet, using nondeterminism, certain states of a minimal DFA are not needed as they correspond to the union of languages of other states. To this end, we distinguish *prime* and *composed* residuals: A residual is called *composed*, if it is the non-trivial union of other residuals. Otherwise, it is called *prime residual language*.

Definition 2.3.14 (Prime and composed residuals)**.** *Let $L \subseteq \Sigma^*$ be a language. A residual $L' \in \mathit{Res}(L)$ is called* composed *if there are $L_1, \ldots, L_n \in \mathit{Res}(L) \setminus \{L'\}$ such that $L' = L_1 \cup \ldots \cup L_n$. Otherwise, it is called* prime*. The set of prime residuals of L is denoted by $\mathit{Primes}(L)$.*

Given a finite state automaton $\mathcal{A} = (Q, Q_0, \delta, F)$ recognizing a regular language L, we call a state $q \in Q$ *prime* if it recognizes a prime residual language of L, i.e., if $L_q \in \mathit{Primes}(L)$.

We can now define the *canonical* RFSA of a regular language. The idea is that its set of states corresponds exactly to its prime residuals. Moreover, the transition function should be *saturated* in the sense that a transition to a state should always exist if it does not change the accepted language. Formally, we define:

Definition 2.3.15 (The canonical RFSA $\mathcal{R}(L)$)**.** *Let L be a regular language. The* canonical RFSA *of L, denoted by $\mathcal{R}(L)$, is the tuple (Q, Q_0, δ, F) where:*

- $Q = \mathit{Primes}(L)$,
- $Q_0 = \{L' \in Q \mid L' \subseteq L\}$,
- $\delta(L_1, a) = \{L_2 \in Q \mid L_2 \subseteq a^{-1}L_1\}$ *for* $L_1 \in Q$ *and* $a \in \Sigma$, *and*
- $F = \{L' \in Q \mid \varepsilon \in L'\}$.

Note that the canonical RFSA of a regular language is well-defined as the set of prime residuals for a regular language is finite and, for each $a \in \Sigma$ and $L' \in \mathit{Res}(L)$, we have $a^{-1}L' \in \mathit{Res}(L)$. Moreover, as shown in [DLT02], we actually have $L(\mathcal{R}(L)) = L$. By definition, there is a single and thus unique canonical RFSA for every regular language. We say that an RFSA $\mathcal{R}$ is *canonical* if it is the canonical RFSA of $L(\mathcal{R})$.

Example 2.3.16. In Figure 2.3 two minimal NFA are depicted, recognizing the regular language $L = L(((a^*|b^*)a\Sigma)^*)$ from Example 2.3.4. Figure 2.3(a) shows an RFSA for L. In Example 2.3.12, we already determined the residuals: $\varepsilon^{-1}L = L_{r_0} = L$, $b^{-1}L = L_{r_1} = L_{p_3}$, and $(ba)^{-1}L = L_{r_2} = L_{p_4}$. These three residuals are the prime residuals of L, as can be shown easily: $L_{p_0} = L_{r_0} \cup L_{r_1}$, $L_{p_1} = L_{r_1} \cup L_{r_2}$, $L_{p_2} = L_{r_0} \cup L_{r_1} \cup L_{r_2}$, $L_{p_3} = L_{r_1}$, and $L_{p_4} = L_{r_2}$.

As all its states correspond to prime residual languages, RFSA $\mathcal{R}_L$ is canonical. However, NFA $\mathcal{R}_l'$ from Figure 2.3(b) is still an RFSA as all its states correspond to residual languages (even to prime residuals) but it is not canonical because its transition set is not saturated. Obviously the NFA from Figure 2.1(b) on page 12 is not an RFSA. The language of state q_3, for example, only contains the empty word, but $\{\varepsilon\}$ is not a residual of L. ◇

2.3.3 Residual Universal Finite-State Automata

In analogy to RFSA we define *residual universal finite-state automata* (RUFA) as automata which—in contrast to NFA which have nondeterministic or-transitions—have deterministic *and*-transitions, and whose states all recognize residual languages.

Definition 2.3.17 (Universal finite-state automaton)**.** *A* universal finite-state automaton *(UFSA, for short) over Σ is a tuple $\mathcal{U} = (Q, Q_0, \delta, F)$, where:*

- *Q is a non-empty finite set of states,*
- *$Q_0 \subseteq Q$ is a set of initial states,*
- *$\delta : Q \times \Sigma \to 2^Q \setminus \{\emptyset\}$ is the transition function, and*
- *$F \subseteq Q$ is a set of final states.*

Syntactically, a UFSA is nothing else than a finite-state automaton. To define the semantics of a UFSA $\mathcal{U}$, we now introduce the finite-state automaton $\mathcal{A}_{\mathcal{U}}$.

Definition 2.3.18 (Semantics of UFSA)**.** *Given a UFSA $\mathcal{U} = (Q, Q_0, \delta, F)$, we define the DFA $\mathcal{A}_{\mathcal{U}} = (Q', Q'_0, \delta', F')$ over Σ, such that:*

- *$Q' = 2^Q$,*
- *$Q'_0 = \{Q_0\}$,*
- *$\delta'(P, a) = \bigcup_{p \in P} \delta(p, a)$ for $P \in Q'$ and $a \in \Sigma$, and*
- *$F' = 2^F$.*

The language recognized by UFSA $\mathcal{U}$ can now be described on basis of $\mathcal{A}_{\mathcal{U}}$: $L(\mathcal{U}) = L(\mathcal{A}_{\mathcal{U}})$.

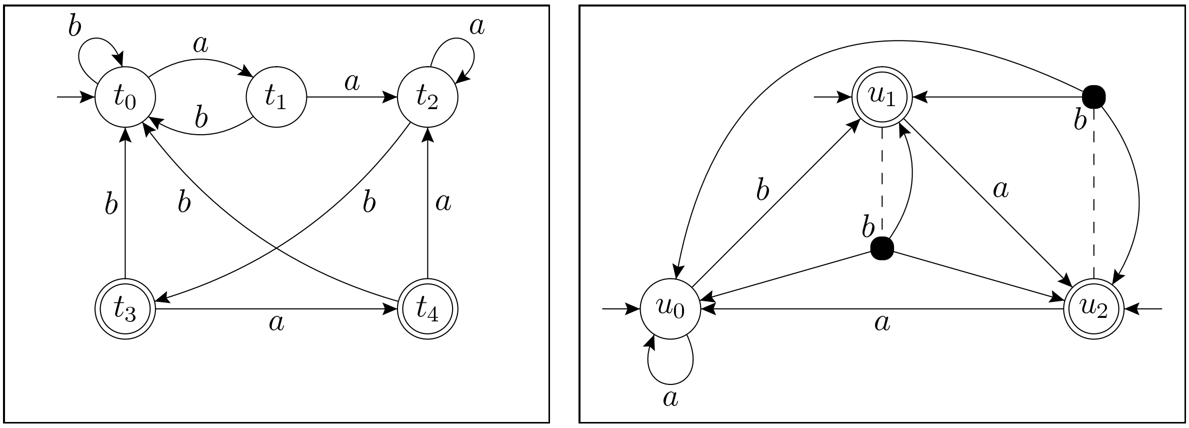
As in the case of NFA and DFA, we extend the transitions function δ to words. Similarly to the finite-state automata we became acquainted with so far, we define residual languages of UFSA. For a state $q \in Q$ of UFSA $\mathcal{U} = (Q, Q_0, \delta, F)$, $L_q = L(\mathcal{A}_{\mathcal{U},q})$, where $\mathcal{A}_{\mathcal{U},q} = (Q, \{q\}, \delta, F)$, i.e., the language accepted by $\mathcal{U}$ if state $\{q\}$ is regarded as initial state.

On basis of this definition, we are now able to derive the notion of residual universal finite-state automata.

Definition 2.3.19 (Residual universal finite-state automaton)**.** *A* residual universal finite-state automaton *(RUFA, for short) is a UFSA $\mathcal{U} = (Q, Q_0, \delta, F)$ for which all states recognize residual languages, i.e., for each $q \in Q$, $L_q \in Res(L(\mathcal{U}))$.*

To derive a canonical version of RUFA, we need to define an analogon to prime and composed states in the setting of RFSA. By abuse of nomenclature, we will call them $\cap$-*prime* and $\cap$-*composed*, as from the context it will always be clear, which definition is being referred to.

As the property of residuals to be prime or composed is a language property, we first define these notions for regular languages L and then use it to define *canonical RUFA*.



(a) Minimal DFA $\mathcal{A}_{L'}$ (b) Canonical RUFA $\mathcal{U}_{L'}$

Figure 2.4: Minimal DFA for a regular language L' and corresponding canonical RUFA

Definition 2.3.20 ($\cap$-prime and $\cap$-composed residuals)**.** *Let $L \subseteq \Sigma^*$ be a language. A residual $L' \in Res(L)$ is called* $\cap$-composed *if there are $L_1, \ldots, L_n \in Res(L) \setminus \{L'\}$ such that $L' = L_1 \cap \ldots \cap L_n$. Otherwise, it is called* $\cap$-prime. *The set of $\cap$-prime residuals of L is denoted by $Primes^{\cap}(L)$.*

In the spirit of RFSA we can now define the notion of *canonical RUFA*. A *canonical RUFA* is a RUFA for which all states correspond to $\cap$-prime residual languages.

Definition 2.3.21 (Canonical RUFA)**.** *Let L be a regular language. Then, the* canonical RUFA *of L, denoted by $\mathcal{U}(L)$, is the tuple (Q, Q_0, δ, F) where:*

- $Q = Primes^{\cap}(L)$,
- $Q_0 = \{L' \in Q \mid L \subseteq L'\}$,
- $\delta(L_1, a) = \{L_2 \in Q \mid a^{-1}L_1 \subseteq L_2\}$ *for* $L_1 \in Q$ *and* $a \in \Sigma$, *and*
- $F = 2^{\{L' \in Q \mid \varepsilon \in L'\}}$.

Like in Example 2.3.16, we could derive the residual languages of the regular language L' defined by the minimal DFA from Figure 2.4(a) and determine the $\cap$-prime residuals of L. As $\cap$-primes only the languages $(aa)^{-1}L$, $(aab)^{-1}L$ and $(aaba)^{-1}L$ remain, which just represent the languages of the state of the canonical RUFA depicted in Figure 2.4(b). The residuals $\varepsilon^{-1}L = (aa)^{-1}L \cap (aab)^{-1}L \cap (aaba)^{-1}L$ and $a^{-1}L = (aa)^{-1}L \cap (aaba)^{-1}L$ are $\cap$-composed. For the corresponding automata $\mathcal{A}_{L'}$ and $\mathcal{U}_{L'}$ thus holds: $L_{t_0} = L_{u_0} \cap L_{u_1} \cap L_{u_2}$, $L_{t_1} = L_{u_0} \cap L_{u_2}$, $L_{t_2} = L_{u_0}$, $L_{t_3} = L_{u_1}$, and $L_{t_4} = L_{u_2}$.

Part I

Inductive Learning Techniques and their Applications

3 A Plethora of Learning Techniques

We are now going to introduce the field of automata inference which will be employed in later chapters for automata and distributed system synthesis.

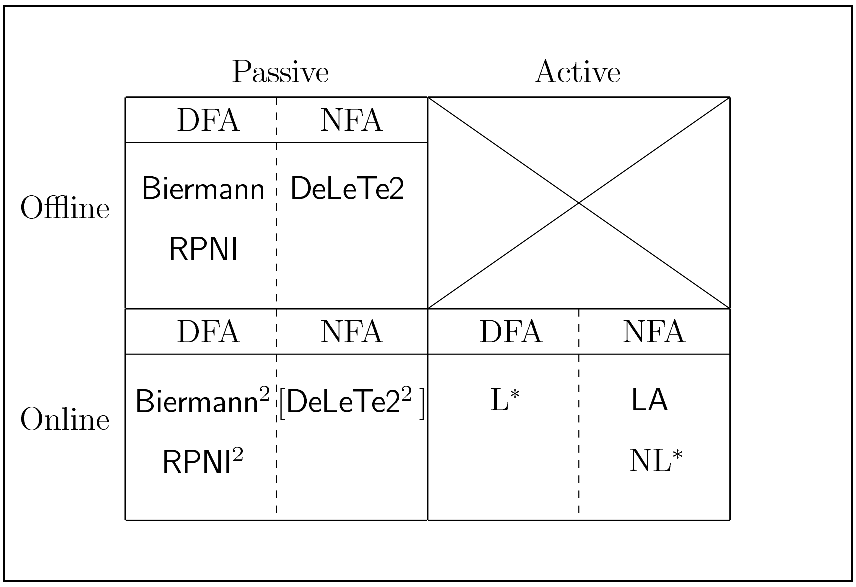
Let us first clarify the term *synthesis*. The word is of Latin origin and in its essence means *composition* or *creation* of something complex from simpler components. We already saw in Chapter 1 that, depending on the context, the goals of synthesis in computer science can be manifold. Synthesis algorithms are employed, e.g., to derive winning strategies for games (cf. [Tho09] for a nicely elaborated survey) for example in the field of controller synthesis [KPP09], to synthesize acceptors from structured data [TB73] (also called *automata synthesis*), or to create abstract models of systems or even code from given information like a requirements document [BKKL09, BCG$^+$99].

In general, synthesis of automata is a broad field. In this thesis, we focus on the latter two fields of application. This chapter is devoted to so-called *inductive inference algorithms* (also called *inference algorithms*, *learning algorithms*, or *learners*). The field of grammatical inference has many areas of application [NP94]. Among them are *pattern recognition* [Fu81, GV90], approaches to *automatic translation* [CGV94], biological systems [CK06], etc. The learning algorithms presented here only get a finite set of classified strings as input, which (to some extent) characterizes the target language. The main task then consists of (semi-) automatically generating a model (also called *hypothesis* or *target*) in terms of a finite-state automaton which treats the input strings exactly according to their classification—this is usually called *consistent* with the input—and generalizes it to obtain a model capable of dealing with all possible inputs.

More precisely, in this chapter we introduce fundamental learning techniques for inferring deterministic and nondeterministic finite-state automata and discuss advantages and disadvantages of the approaches wrt. the context in which we will employ learning in later chapters. Note that with this chapter we do not intend to give a perfectly sophisticated and complete description of all kinds of learning algorithms but to introduce the main ideas behind some interesting representatives. A nicely elaborated overview over the whole field of grammatical inference subsuming the field of inductive inference that is tackled in this thesis is given by de la Higuera in [dlH05]. A more detailed survey of several inference algorithms, their variations, and applications in the area of verification can be found in [Leu07].

3.1 The Big Picture

The main aim of learning algorithms in the field of automata synthesis is to (uniquely) detect a representative automaton that accepts a given set of words over an alphabet Σ. Sometimes this set is enriched by a set of negative words, which must not be accepted. Typically, learning algorithms are divided into four categories, namely *passive* and *active* algorithms, and *offline* and *online* algorithms according to their capabilities and access to information about the target language. Even if the terms active and online, and passive and offline are often regarded as synonyms, there is a slight difference, which we will



	Passive		Active	
	DFA	NFA	DFA	NFA
Offline	Biermann RPNI	DeLeTe2		
Online	Biermann^2 RPNI^2	[DeLeTe2^2]	L*	LA NL*

Figure 3.1: Overview over classes of learning algorithms

briefly address. According to this distinction, we will classify all forthcoming learning algorithms of this chapter.

Active learning algorithms are allowed to autonomously ask queries to some kind of information source—e.g., a human user—i.e., they may ask for the membership of certain words wrt. the target language. Thus, this class of algorithms does not retrieve an arbitrary representation of the target language but has active impact on the set of examples that may be used for deriving a target automaton. In contrast, *passive learning algorithms* are only passively provided with examples and cannot or may not ask questions regarding language membership. They totally depend on data presented by some kind of limited or unlimited stream of classified words. *Online learning algorithms* work incrementally by successively enhancing the hypothesis whereas *offline learning algorithms* only proceed on one a priori given sample to derive one possible hypothesis which respects the input.

Combinations of the properties passive, active and offline, online are possible. In Subsection 3.2.1, e.g., we introduce a passive offline learning algorithm called RPNI. In [Dup96a] this algorithm is extended to an incremental version called RPNI^2, yielding a passive online learning algorithm (where, e.g., examples can be provided via a stream of incoming words on which the learning algorithm—as mentioned before—has no influence).

In the literature, learning algorithms based on fixed given data and continuously streamed data, respectively, as sources of information, are sometimes called *learning from given data* and *learning from sequential data*, respectively. Dupont showed in his thesis [Dup96b] that these notions are equivalent.

More examples for and differences between these classes will be addressed in the subsequent sections. Note, however, that there is no class of *active offline* learning algorithms because the definitions of active and offline are mutually exclusive. Offline algorithms only work on an a priori fixed set of examples, i.e., it cannot be extended as it would have to be the case for active algorithms.

Besides this distinction into passive/active and offline/online algorithms, we subdivide the algorithms of these classes into the ones that infer DFA and the ones that learn NFA. Figure 3.1 gives a clearly arranged overview over the prementioned classes and the classification of the inference algorithms described or mentioned in this chapter. The crossed lines in the upper right quadrant signify that the class of active offline learning algorithms does not exist (according to the definition of active and offline given before).

Additionally, we would like to differentiate between the following two problems:

($Q1$) Given a set S of classified examples, propose some kind of explanation, i.e., some (minimal) automaton which is capable of classifying the examples correctly (later called *consistently*), i.e., in compliance with S.

($Q2$) Having a target language L in mind, derive a minimal (in some sense that has to be made precise in the particular setting) model which exactly recognizes the target L.

Before we go into details about the concrete instances of the classes mentioned above, let us, as a short reminder, briefly mention the most important theoretical results from the preliminaries in the context of learning. The formal basis for all algorithms intending to learn regular languages is due to Myhill and Nerode stating that for each regular language L there exists a unique minimal DFA recognizing L (cf. Theorem 2.3.3). Moreover, *Nerode's right congruence* $\sim_L$ is an important characterization of regular languages.

It is known from Definition 2.2.4 that a language L is regular if and only if the right-congruence relation $\sim_L$ has finite index, i.e., if L contains finitely many equivalence classes. These equivalence classes correspond to the states in the canonical DFA $\mathcal{A}_L$ recognizing the regular language L.

As we will see in the next sections, these properties of regular languages are usually exploited in learning algorithms for regular languages.

Let us, for the rest of this chapter, denote $\mathcal{A}_L$ the minimal DFA for a regular language $L \subseteq \Sigma^*$.

3.1.1 A Formal Classification of Learnability

The first known approach for formally defining the notion of *learnability* and attempt to study the problem of automata inference was undertaken by Gold [Gol67, Gol78] under the name *identification in the limit*. It can be seen as a theoretical framework for analyzing and comparing different learning approaches.

Definition 3.1.1 (Identification in the limit)**.** *Formally, a learning algorithm* $\mathfrak{A}$ identifies in the limit *a class of languages* $\mathfrak{L}$ *by means of hypotheses from the hypothesis space* $\mathfrak{H}$ *iff* $\forall L \in \mathfrak{L}$ *the infinite sequence* $h_1, h_2, \ldots$ *of hypotheses output by* $\mathfrak{A}$ *converges to a hypothesis* $h \in \mathfrak{H}$ *such that* $L(h) = L$.

Thus, the learning procedure is perceived as an infinite process of (1) being presented an element of the language to infer and (2) deriving a hypothesis on basis of all data received in the past. Thereby, after a finite number of wrong hypotheses, any (regular) language can be identified in the limit by a correct hypothesis. Hence, there exists $k' \in \mathbb{N}$ and a sequence of hypotheses $h_1, h_2, \ldots$ such that for all $k < k'$: $L(h_k) \neq L$ but for all $k \geq k'$: $L(h_k) = L$. Note however, that the procedure mentioned before is not necessarily able to detect the correctness of a hypothesis h, i.e., whether $L(h) = L$.

Thus, the above notion can be seen as a kind of *quality* criterion for classes of languages that tells us whether or not a class of languages can be learned at all. Another interesting question arising in this context is if this class of languages is efficiently learnable. De la Higuera extended the notion of "identifiability in the limit" in [dlH97] to also take into account the *complexity* of learning algorithms and the classes of acceptors that are to be learned. A *representation class* is a class of acceptors that are capable of recognizing certain language classes. In our case, these representation classes are classes of automata, like DFA, NFA, RFSA, etc, accepting regular languages. In different settings also context-free grammars or Turing machines could be representation classes. An element of a representation class is sometimes called a *representation*.

Definition 3.1.2 (Identification in the limit from polynomial time and data [dlH97])**.** *A representation class $\mathcal{R}$ is* identifiable in the limit from polynomial time and data *iff there exist two polynomials p and q and an inference algorithm $\mathfrak{A}$ such that the following two properties hold:*

(i) $\mathfrak{A}$ learns a representation $R \in \mathcal{R}$ from input data (a set S_+ of positive words and a set S_- of negative words) of size $m = |S_+| + |S_-|$ in time $\mathcal{O}(p(m))$.

(ii) For each representation $R \in \mathcal{R}$ of size n, there exists a set of input data (S'_+, S'_-) of size at most $q(n)$ (with $S'_+ \subseteq S_+$ and $S'_- \subseteq S_-$) such that $\mathfrak{A}$ learns a representation R' equivalent to R.

Note: the size of the input (S_+, S_-) is defined to be the sum of the length of the elements contained in $S_+ \cup S_-$. An inference algorithm fulfilling the properties above, is called a polynomial learner*.*

In [Gol78] Gold showed that the class of regular languages is identifiable in the limit from polynomial time and data using DFA as representation, and in [dlH97] de la Higuera proved a negative result for the class of NFA.

Theorem 3.1.3 (Identification in the limit of DFA and NFA [Gol78, dlH97])**.**

- *The representation class of DFA is identifiable in the limit from polynomial time and data (using the class of DFA as representation).*
- *The representation class of NFA is not identifiable in the limit from polynomial time and data (using the class of NFA as representation).*

Let MinDFA be the decision problem that given a natural number n decides whether there is a DFA with less than n states being consistent with the input data.

Theorem 3.1.4 ([Gol78])**.** *The decision problem MinDFA is NP-complete.*

3.2 Offline Learning

The idea of *offline learning algorithms* is that, provided with an initial set of positive and negative examples, one possible automaton (also called hypothesis automaton) has to be derived which accepts all given positive and rejects all given negative strings. The algorithm is neither presented any new examples extending the initial set, nor allowed to ask any further questions. Therefore, it has to restrict to the given set of words as the only source of information. The offline algorithms we regard in this section can be subdivided into ones that infer DFA and ones that learn NFA. We first describe two algorithms learning DFA in a passive offline fashion, followed by a subsequent subsection presenting a passive offline NFA learning algorithm. Note that all offline learning algorithms are passive. Nevertheless, as shortly mentioned before, there are approaches for turning several of these passive offline learning algorithms into passive online learning algorithms.

3.2.1 Learning Deterministic Automata

In this subsection, we will detail on two inference algorithms implementing the offline learning paradigm for DFA, described above.

Biermann

One of the initial offline learning approaches was undertaken by Biermann and Feldman [BF72] and will subsequently be called the Biermann algorithm. In its original version it receives as input a fixed set of positively classified strings S_+ and negatively classified strings S_- and has to output a DFA consistent with the sample.

To ease presentation, we extend the input collection of positive and negative strings $S = (S_+, S_-)$ (also called *sample*) to a set $\widehat{S} = (S_+, S_-, S_?)$, called *extended sample*, where $S_?$ contains all prefixes of $S_+ \cup S_-$ that are not contained in the union $S_+ \cup S_-$. Note that the extended sample is only used internally. For a finite set of words, the prefix tree acceptor is the DFA that exactly accepts all words from S_+ (cf. Definition 2.3.10). Therefore, its size is an upper bound for the size of the minimal DFA we want to infer. Thus, intuitively, the extension of S to $\widehat{S}$ helps to create the maximal state space possible for inferring the hypothesis DFA. However, as we do not care for the classification of these additional strings, we assign them the new classification "?" called *unknown* or *don't-care*. Let $|\widehat{S}| = |S_+| + |S_-| + |S_?|$. Moreover, by abuse of notation, we define a partial function S with finite, prefix-closed domain $\mathcal{D}(S)$ by: $S : \Sigma^* \to \{+, -, ?\}$ which determines the classification of each element of $\widehat{S}$. Hence,

$$S(u) = \begin{cases} + \text{, if } u \in S_+ \\ - \text{, if } u \in S_- \\ ? \text{, if } u \in \mathit{pref}(S_+ \cup S_-) \setminus (S_+ \cup S_-). \end{cases}$$

Definition 3.2.1 (Consistency). *A finite-state automaton $\mathcal{A}$ is called* consistent *with an extended sample $\widehat{S} = (S_+, S_-, S_?)$ iff:*

$$u \in \mathcal{D}(S) \Rightarrow [S(u) = + \Rightarrow u \in L(\mathcal{A}) \wedge S(u) = - \Rightarrow u \notin L(\mathcal{A})]$$

The objective now is to come up with a hypothesis DFA (of minimal size in its class) being consistent with the set $\widehat{S}$. This corresponds to question $(Q1)$ from Section 3.1 on page 21.

For DFA $\mathcal{A}$ and word $u \in \Sigma^*$ let q_u be the unique state that $\mathcal{A}$ is in after reading word u. The crucial idea now is—because the hypothesis is not derived yet—to regard the q_u as variables (over states) and establish a set of constraints for the assignments of such variables. These constraints intend to represent Nerode's right congruence. Let the set of equations for a constraint satisfaction problem (CSP, for short) over the extended sample $\widehat{S}$ be:

$$\begin{aligned} \mathit{CSP}(\widehat{S}) := \quad & \{q_u \neq q_{u'} \mid S(u) = + \text{ and } S(u') = -\text{, or vice versa}\} && (\mathit{CSP}_1) \\ \cup \quad & \{q_u = q_{u'} \Rightarrow q_{ua} = q_{u'a} \mid a \in \Sigma, ua, u'a \in \mathcal{D}(S)\} && (\mathit{CSP}_2) \end{aligned}$$

In this definition, CSP_1 assures that final and non-final states are distinguished (i.e., they feature different classifications on the empty suffix) and CSP_2 guarantees that the transition function is deterministic. An *assignment* of $\mathit{CSP}(\widehat{S})$ is a function $\beta : \mathcal{D}(\mathit{CSP}(\widehat{S})) \to \mathbb{N}$. We call β a *solution* for the constraint satisfaction problem $\mathit{CSP}(\widehat{S})$ if it

```
BIERMANN($\Sigma, S_+, S_-$):
 1 $\widehat{S} = (S_+, S_-, S_?)$;
 2 for $i = 1, \ldots, N$ where $N = |\widehat{S}|$   // test if there is a solution with $i$ variables
 3   do
 4       Retrieve an assignment $\beta : \mathcal{D}(CSP(\widehat{S})) \to \{1, \ldots, i\}$ for the variables in $\mathcal{D}(CSP(\widehat{S}))$;
 5       Test whether $\beta \models CSP_1 \wedge CSP_2$;
 6       if Test succeeds
 7         then
 8             $Q = \{1, \ldots, i\}$;
 9             $q_0 = q_\varepsilon$;
10             Calculate function $\delta$ s.t. $\delta(m, a) = \{m'\}$ if there are $q_u$ and $q_{ua} \in \mathcal{D}(CSP(\widehat{S}))$
11                        s.t. $q_u = m$ and $q_{ua} = m'$;
12             Calculate set $F$ s.t. for $q_u \in \mathcal{D}(CSP(\widehat{S}))$ with $q_u = m$ :
13                        $u \in S_+ \Rightarrow m \in F$ and
14                        $u \in S_- \Rightarrow m \notin F$;
15             break;
16 return $\mathcal{H} = (Q, \{q_0\}, \delta, F)$;
```

Table 3.1: Biermann: (passive) offline learning algorithm for inferring minimal DFA

satisfies the equations from CSP_1 and CSP_2 over the natural numbers ($\beta \models CSP_1 \wedge CSP_2$, for short).

In Table 3.1 the pseudocode of the Biermann learning approach is presented. To resolve the nondeterminism (in line 4), SAT solvers may be employed. Though the SAT problem for Boolean formulas is NP complete, there have been tremendous improvements regarding SAT solving in the last few years [BBH06]. A solution to our constraint satisfaction problem $CSP(\widehat{S})$ can now be derived by transforming it to a satisfiability problem and employing off-the-shelf SAT solvers (e.g., `RSat`, `MiniSAT`, `SAT4J`, `Spear`, etc.) to solve the SAT instance, and finally retrieve a result for the CSP. A detailed description of this transformation can be found in [GLP06].

As the following theorem states, if such a solution exists for $CSP(\widehat{S})$, we can derive a hypothesis automaton $\mathcal{H}$ which is consistent with the extended sample $\widehat{S} = (S_+, S_-, S_?)$.

Theorem 3.2.2 (Biermann: Correctness [BF72])**.** *For an input sample $S = (S_+, S_-)$, there exists a DFA $\mathcal{H}$ with n states which is consistent with $\widehat{S}$ iff $CSP(\widehat{S})$ is solvable over $\{1, \ldots, n\}$. Moreover, the hypothesis is of the form $\mathcal{H} = (Q, \{q_0\}, \delta, F)$, where:*

- *$Q = \{1, \ldots, n\}$,*
- *$q_0 = q_\varepsilon \in Q$,*
- *$\delta : Q \times \Sigma \to 2^Q$ is a function satisfying $\delta(m, a) = \{m'\}$ ($m \in Q$, $a \in \Sigma$), if there are q_u and $q_{ua} \in \mathcal{D}(CSP(\widehat{S}))$ fulfilling $q_u = m$ and $q_{ua} = m'$.*
- *$F \subseteq Q$ is a set satisfying: for $q_u \in \mathcal{D}(CSP(\widehat{S}))$ with $u \in S_+$ implies $q_u \in F$, and $u \in S_-$ implies $q_u \notin F$.*

Unlike other learning algorithms—as for example Angluin's L* algorithm, which gives an answer to question ($Q2$) from Section 3.1 and will be presented in Section 3.3—Biermann's approach gives a solution to question ($Q1$) and, hence, does not necessarily yield a smallest automaton being consistent with $\widehat{S}$. This shortcoming can however be resolved by performing the learning task described above in an iterated fashion. In Table 3.1 this approach is presented in pseudocode. We now describe this iterative procedure in more detail.

In order to obtain a DFA with a minimal number of states being consistent with $\widehat{S}$ using Biermann's approach, we have to find the correct value $n^* \in \mathbb{N}$ such that for $n = n^*$ there is a solution of $CSP(\widehat{S})$, but for any $n' < n^*$ no solution exists. This goal can be achieved by repeatedly increasing the value of n of Theorem 3.2.2 (starting from $n = 1$) and checking whether or not there exists a solution to $CSP(\widehat{S})$ over $\{1, \ldots, n\}$, i.e., if there is a minimal DFA consistent with $\widehat{S}$ which has n states. As there is an a priori given upper bound N on the number of states of the hypothesis automaton $\mathcal{H}$, this approach will always terminate yielding a correct minimal DFA being consistent with $\widehat{S}$. A trivial upper bound is given by $N = |\widehat{S}|$ (or even $N = |S_+| + |S_?|$, i.e., the number of states of the prefix tree acceptor).

As the upper bound on the number of states is known a priori, however, it could be an improvement to employ *binary search* within the algorithm of Table 3.1 to speed up the inference of $\mathcal{H}$. If the upper bound was N, the algorithm started with the value $\frac{N}{2}$. If no solution could be calculated for this setting, we had to increase our bound to $\frac{3n}{4}$ or in case there was a solution, store it and try to achieve one with a new bound of $\frac{N}{4}$.

Theorem 3.2.3 (Biermann: Complexity)**.** *Our iterative implementation of the* Biermann *learning algorithm has an exponential worst-case time complexity (in the number of states of the target automaton).*

Proof: To find a smallest minimal DFA of size n consistent with the input, with this version of the Biermann algorithm we have to solve $\log(n)$-times (using binary search) an NP complete problem of size $\mathcal{O}(n)$. This overall yields an algorithm which runs in $\mathcal{O}(\log(n) \cdot 2^n)$ time. Note that this result is in line with the theorem by Gold (cf. Theorem 3.1.4). ☐

We mentioned already that many passive offline approaches can also be realized in a passive online manner. The Biermann algorithm is such an example. In Figure 3.1 we called it Biermann^2. In its new version, the learning algorithm is continuously confronted with new classified strings from a stream and can converge to an a priori fixed regular language L. Note, however, that with this ability the algorithm would need exponentially (in the size of the minimal DFA $\mathcal{A}_L$ for language L) many input words to infer $\mathcal{A}_L$. For each additional word the algorithm would be invoked which had to solve a SAT instance (for iteratively increasing size), yielding an exponential algorithm and thus making this version of the algorithm infeasible for solving Question ($Q2$) from Section 3.1.

Regular Positive and Negative Inference (RPNI)

We are now introducing the *regular positive and negative inference* (RPNI, for short) algorithm which is a passive offline learning algorithm for inferring DFA.

Again, we call a tuple $S = (S_+, S_-)$ with a set $S_+ \subseteq \Sigma^*$ of positive words and a set $S_- \subseteq \Sigma^*$ of negative words a *sample*. For $S'_+ \supseteq S_+$ and $S'_- \supseteq S_-$, $S' = (S'_+, S'_-)$ is called an *extension* of S. Being provided with a finite set of positive strings $S_+ \subseteq \Sigma^*$ and negative strings $S_- \subseteq \Sigma^*$ such that $S = (S_+, S_-)$, question ($Q1$) from Section 3.1 has to be solved, i.e., a DFA $\mathcal{A}$ has to be learned such that $L(\mathcal{A})$ is *consistent* with S which means that for all $w \in S_+$, $w \in L(\mathcal{A})$ and for all $w \in S_-$, $w \notin L(\mathcal{A})$. The underlying idea of the RPNI algorithm is to first build the prefix automaton $\mathcal{A}(S_+)$ (cf. Definition 2.3.10)

```
RPNI ($S_+, S_-$):
Generate prefix tree acceptor $\mathcal{A}(S_+) = (Q_+, \{q_0^+\}, \delta_+, F_+)$;
Let $Q_+ = \{u_0, \ldots, u_n\}$ be in canonical order;
$\sim_0 := \{(u, u) \mid u \in Q_+\}$;
for $i = 1$ to $n$    // for all states
    do
        // check equivalence to all smaller (i.e., wrt. $<_{\text{lex}}$) states
        if $u_i \not\sim_{i-1} u_j$ for all $0 \leq j \leq i-1$ then
          $j := -1$;
          repeat
                $j := j + 1$;
                $\sim :=$ smallest congruence with $\sim_{i-1} \subseteq \sim$ and $u_i \sim u_j$;
                $B := \mathcal{A}(S_+)_{/\sim}$;
          until $L(B) \cap S_- = \emptyset$;  // current hypothesis consistent with $S_-$
          $\sim_i := \sim$;
        else $\sim_i := \sim_{i-1}$;
return $\mathcal{H} = \mathcal{A}(S_+)_{/\sim_n}$
```

Table 3.2: RPNI: (passive) offline learning algorithm for inferring (minimal) DFA

accepting exactly the words from S_+ and then use S_- to perform state-merging operations on $\mathcal{A}(S_+)$. As there are exponentially many possibilities for such operations, we have to fix some order to obtain an efficient algorithm. Hence, we define a canonical order (length-lexicographical order, cf. Definition 2.2.2) on the set of states of $\mathcal{A}(S_+)$ to which merging is performed. Subsequently, the algorithm executes a search through the current set of states of automaton $\mathcal{A}(S_+)$ according to the canonical order and merges states whenever they cannot be distinguished by the algorithm on basis of the given sample. Whenever this procedure terminates on input S, a DFA consistent with sample S is output.

Note, however, that RPNI does not necessarily yield minimal DFA. To derive a minimal DFA, the sample S has to be *complete*. Intuitively, S is complete for a regular language L, if S provides enough information to uniquely determine a DFA for the target language L. I.e., it has to contain sufficient information to distinguish non-equivalent states, determine the transition function, and detect final states. It can be shown that such a complete sample exists for each regular language L. The set of all complete samples for a given regular language L accepted by a DFA $\mathcal{A}_S$ which is consistent with S has the following properties:

- S is a complete sample for $L \Longrightarrow L(\mathcal{A}_S) = L$,
- there is a complete sample for L that is of size polynomial in the size of $\mathcal{A}_S$,
- every extension S' of S is a complete sample for L.

We now introduce two auxiliary notions before we can formally define the concept of a complete sample for a regular language L.

Definition 3.2.4 (Shortest prefixes and kernel)**.** *For regular language $L \subseteq \Sigma^*$, let:*

$$\begin{aligned} SP(L) &= \{w \in \mathit{pref}(L) \mid \forall u \in [w]_{\sim_L} : w <_{\text{lex}} u\} \\ K(L) &= \{w \in \mathit{pref}(L) \mid \exists u \in SP(L),\ a \in \Sigma :\ w = ua\} \end{aligned}$$

Here, $\sim_L$ represents the Nerode right congruence. The abbreviation *SP* stands for *shortest prefixes*. The set $SP(L)$ contains *minimal representative* words which, intuitively speaking, correspond to the states in the minimal DFA for L. More precisely, the set $SP(L)$ contains the shortest strings (wrt. $<_{\text{lex}}$) each reaching a different state of the minimal DFA $\mathcal{A}_L = (Q, Q_0, \delta, F)$. Hence, $|SP(L)| = |Q|$. The set $K(L)$ is called the *kernel* of L and represents the transitions of $\mathcal{A}_L$. The kernel contains a word for every transition in $\mathcal{A}_L$.

Definition 3.2.5 (Complete sample). *A sample* $S = (S_+, S_-)$ *is called* complete *for* L*, if the following properties hold:*

(i) $\forall w \in L. \exists u \in S_+ : u \sim_L w$ *(final states are covered),*

(ii) $\forall w \in K(L). \exists u \in \Sigma^* : wu \in S_+$ *(essential transitions are covered),*

(iii) $\forall u \in SP(L). \forall v \in K(L)$ *s.t.* $u \not\sim_L v : \exists w \in \Sigma^* :$
$uw, vw \in S_+ \cup S_-$ *and* $uw \in S_+ \iff vw \in S_-$ *($\sim_L$-classes are distinguished).*

Condition *(i)* assures that the set of positive examples covers the set of final states, i.e., for each word in the language we have a representative word in S_+ and the representatives altogether characterize the set of final states. Condition *(ii)* guarantees that all transitions are covered by sample S_+, i.e., the information about the essential transitions in the sample is sufficient to infer a minimal DFA, and condition *(iii)* ensures that using the sample we are able to separate non-equivalent $\sim_L$-classes.

Together with this definition, we obtain the following results:

Theorem 3.2.6 (RPNI: Identification in the limit [OG92]). *The RPNI algorithm identifies the class of regular languages in the limit (using DFA as representation).*

Theorem 3.2.7 (RPNI: Correctness [OG92]). *Provided with an input sample* $S = (S_+, S_-)$ *that is complete for regular language* L*, the RPNI*(S_+, S_-) *algorithm is capable of inferring the minimal DFA* $\mathcal{A}(S_+)_{/\sim_n}$ *accepting* L*.*

Theorem 3.2.8 (RPNI: Complexity [OG92]). *Provided with an input sample* $S = (S_+, S_-)$*, the RPNI algorithm runs in time* $\mathcal{O}(l \cdot |\Sigma| \cdot k^4)$*, where* l *is the sum of the length of all elements from* S_-*, i.e.,* $l = \sum_{w \in S_-} |w|$*,* $|\Sigma|$ *is the size of the alphabet, and* $k = |Q_+|$ *is the number of states of the prefix tree acceptor* $\mathcal{A}(S_+)$*.*

In Table 3.2 a pseudocode implementation of the RPNI algorithm is given.

Actually the RPNI algorithm can be regarded as an improvement of a very similar approach by Trakhtenbrot and Barzdin [TB73]. Their approach also exhibits the following major steps: first, build prefix tree acceptor, second, merge equivalent states, third, if the given set was *characteristic* (i.e., complete) for a regular language L, the minimal DFA for the target language is being deduced. RPNI, however, gets along with a much smaller input sample. Trakhtenbrot et al. require the sample S to be n-complete, meaning that it has to contain all words from $\Sigma^{\leq n} = \bigcup_{i=0,\ldots,n} \Sigma^i$. As the size of $\Sigma^{\leq n}$ is exponential in n, it thus needs an exponential number of classified examples in order to derive a DFA. Another difference is that in RPNI the order in which states are merged is of importance whereas in Trakhtenbrot et al. this is not the case.

The RPNI algorithm is a classical example for a passive offline learning algorithm. Like all passive offline algorithms, RPNI suffers from the drawback that, if new learning data becomes available over time, the algorithm has to start all over again on basis of the augmented sample in order to derive a new hypothesis. This inconvenience can be circumvented, as mentioned in the introduction of this chapter, by extending it to an incremental version in which the learning algorithm is connected to a continuous (random) stream of classified examples—called *sequential presentation*—yielding a passive online version of RPNI. This extension can be found in [Dup96a] under the name RPNI2, which stands for *regular positive and negative incremental inference*. To obtain consistent notation, in Figure 3.1 on page 22 the passive online version of the algorithm is called RPNI^2. It was shown by Dupont [Dup96b] that the class of regular languages is learnable in the limit using his RPNI^2 approach.

3.2.2 Learning Nondeterministic Automata

Now, we focus on inferring regular languages using NFA as representation in an offline manner. As NFA can be exponentially smaller than equivalent minimal DFA, they are often a better means to describe regular languages in a compact way.

DeLeTe2: Learning RFSA Using the RPNI Approach

One of the first attempts to infer regular languages in terms NFA by means of learning algorithms was accomplished by Denis et al. In their paper [DLT04] they extend the RPNI approach, described in the previous subsection, towards learning of RFSA (cf. Definition 2.3.13). Their passive offline learning algorithm DeLeTe2 aims at answering question ($Q1$). Though similar in nature, the DeLeTe2 algorithm goes one step further than its relative RPNI. Instead of checking for language equivalence, DeLeTe2 checks for language inclusion of the states of the current hypothesis.

As in RPNI, the sample set has to meet certain criteria to derive the desired automata. Though addressing similar aspects, these criteria differ from the RPNI approach. Let L be a regular language, $\mathcal{A}_L = (Q, \{q_0\}, \delta, F)$, and, for every $q \in Q$, u_q be the smallest word (according to $<_{\text{lex}}$) which reaches q starting from the initial state of $\mathcal{A}_L$, i.e., $\delta(q_0, u_q) = \{q\}$. Moreover, in DFA $\mathcal{A}_L$ we call a state p smaller than a state q (written $p \leq q$) iff $u_p \leq_{\text{lex}} u_q$. In analogy to the RPNI notion, we define the shortest prefixes and kernel in the setting of RFSA inference:

Definition 3.2.9 (Shortest prefixes and kernel[DLT04]). *Let $L \subseteq \Sigma^*$ be a regular language, $p^* \in Primes(L)$ be the greatest prime of L, i.e., for all $p \in Primes(L) : p \leq p^*$, $\mathcal{A}_L = (Q, Q_0, \delta, F)$, and define:*

$$\begin{array}{lcll} SP(L) & = & \{u_q \in \Sigma^* \mid q \leq p^*\} & \text{to be the set of shortest prefixes of } L\text{, and} \\ K(L) & = & \{u_q a \in \Sigma^* \mid q \leq p^*, a \in \Sigma, \delta(q,a) \neq \emptyset\} & \text{to be the kernel of } L. \end{array}$$

Intuitively, $SP(L)$ contains at least all minimal words classifying the prime states of L and, hence, enough information to determine the states of the target automaton $\mathcal{A}_L$. Moreover, the set $K(L)$ exhibits enough information for constructing transitions.

In addition, we define the partial order $\prec \subseteq \Sigma^* \times \Sigma^*$ over words such that $u \prec v$ if there is no $w \in \Sigma^*$ such that $uw \in S_+$ and $vw \in S_-$. Additionally, $u \simeq v$ if $u \prec v$ and $v \prec u$.

We are now ready to define the notion of *complete samples* for RFSA.

Definition 3.2.10 (Complete sample for inclusion relations [DLT04]). *Let L be a regular language. A sample $S = (S_+, S_-)$ is called* complete for inclusion relations *of L iff the following properties are satisfied:*

```
DeLeTe2 $(S_+, S_-, \Sigma)$:
Initialize $\mathcal{H} = (\emptyset, \emptyset, \emptyset, \emptyset)$;
Let $Pref = pref(S_+) = \{u_0, \ldots, u_n\}$; // ordered in length-lexicographic order
$u = \varepsilon$;
while ($u$ is defined or $\mathcal{H}$ is not consistent with $(S_+, S_-)$)
        do
            if there is $u' \in Q$ such that $u \simeq u'$
              then // $u$ is equivalent to an already existing state $u'$
                   remove $u\Sigma^*$ from $Pref$;
              else // add new state $u$ to $Q$ and add corresponding transitions
                   $Q = Q \cup \{u\}$;
                   if $u \prec \varepsilon$
                     then // $u$ is an initial state
                          $Q_0 = Q_0 \cup \{u\}$;
                   if $u \in S_+$
                     then // $u$ is a final state
                          $F = F \cup \{u\}$;
                   $\delta = \delta \cup \{u' \xrightarrow{x} u \mid u' \in Q, u'x \in Pref, u \prec u'x\}$
                          $\cup \{u \xrightarrow{x} u' \mid u' \in Q, ux \in Pref, u' \prec ux\}$;
                   $\mathcal{H} = (Q, Q_0, \delta, F)$
            Let $u$ be the next word in $Pref$; // if there is none, $u$ is undefined
return $\mathcal{H}$;
```

Table 3.3: DeLeTe2: (passive) offline learning algorithm for inferring (canonical) RFSA

(i) $\forall u \in SP(L) \cup K(L). u \in pref(S_+)$ *(states and transitions are covered),*

(ii) $SP(L) \cap L \subseteq S_+$ *(final states are covered),*

(iii) $\forall u \in SP(L), \forall v \in SP(L) \cup K(L). u^{-1}L \not\subseteq v^{-1}L \Rightarrow \exists w \in \Sigma^*. uw \in S_+$ *and* $vw \in S_-$ *(equivalence classes are distinguished).*

Property (i) addresses coverage of states and transitions. Sample S must contain sufficient information to completely describe the set of states and transitions of the target automaton. As described above, $SP(L)$ contains the information about prime states, i.e., states which have to be contained in the target automaton to yield an RFSA, and $K(L)$ features the corresponding information about transitions. Condition (ii) guarantees that the set of positive strings S_+ contains all minimal words that lead to final (prime) states, i.e., the set of final states is covered by the sample. Finally, item (iii) ensures that residual languages of L that are not in an inclusion relation can be separated, i.e., that equivalence classes of the language L can be distinguished.

In [DLT04], Denis et al. show that there is always a complete sample which has size at most $\mathcal{O}(n^5)$, where n is the size of the minimal DFA $\mathcal{A}_L$.

Table 3.3 exhibits a pseudocode implementation of the DeLeTe2 algorithm which takes as input a sample $S = (S_+, S_-)$ and the alphabet Σ. In contrast to the RPNI algorithm where we start from the prefix tree acceptor and successively merge states yielding smaller and smaller intermediate results, the DeLeTe2 algorithm for inferring regular languages with RFSA as representations works in a bottom-up, iterative fashion. It starts with the empty automaton and gradually adds states (from $pref(S_+)$) that are non-equivalent to the previously added states. Transitions are derived from inclusion relations between the already existing states and the newly added one. The algorithm terminates if the hypothesis is consistent with sample set S or, after all prefixes of S_+ have been considered.

Thus, DeLeTe2 eventually returns a hypothesis automaton $\mathcal{H}$ which is an RFSA recognizing L if the specified sample set S was complete for inclusion relations for a regular language L. This is, unfortunately, not optimal in the sense that, in general, the canonical RFSA cannot be derived by this algorithm. Regarding the size of the RFSA learned by DeLeTe2, the automata are located between the canonical RFSA and the minimal DFA. In Chapter 4, we will present a new active online learning approach which is capable to infer the canonical RFSA for any regular language L.

A more serious drawback of DeLeTe2, however, is that, if the input sample is not complete, the target automaton can, in general, not be guaranteed to be consistent with the input $S = (S_+, S_-)$ and, moreover, the hypothesis needs not be an RFSA. Note that in contrast the RPNI algorithm guarantees the DFA to be consistent with the sample even if the sample was not complete for regular language L (and thus the derived DFA not minimal for L). Denis et al. propose two ways of tackling this limitation. The obvious but inadequate fix would be to return the prefix tree acceptor in case the hypothesis is not consistent with sample S. The other, more elaborate possibility is to proceed as follows: as the relation $\prec$ is an approximation of the real inclusion relation of residual languages, which indeed is transitive and right-invariant for concatenation (i.e., if $u^{-1}L \subseteq v^{-1}L$ then for all $w \in \Sigma^*$: $(uw)^{-1}L \subseteq (vw)^{-1}L$), in every step—when testing the relation $p \prec q$ between two states p, q—the extended algorithm completes the relation by adding missing elements to be transitive and right-invariant. New transitions are added according to the augmented $\prec$-relation. If the resulting language is consistent with the input the algorithm continues and in case consistency could not be achieved the original relation $p \prec q$ and all elements added thereafter are considered invalid and removed from $\prec$. The extended algorithm always returns an automaton that is consistent with the input sample.

Theorem 3.2.11 (DeLeTe2: Identification in the limit [DLT04])**.** *The class of regular languages over Σ:*

- *is not identifiable in the limit from polynomial time and data using NFA or RFSA as representation (unless $|\Sigma| = 1$).*
- *is identifiable in the limit from polynomial time and data in terms of RFSA if DFA are used as representation.*

Theorem 3.2.12 (DeLeTe2: Correctness [DLT04])**.** *Provided a complete sample $S = (S_+, S_-)$ and an alphabet Σ, the DeLeTe2 algorithm eventually outputs the saturated subautomaton $\mathcal{A}_{u_p}$ of L (i.e., an RFSA whose size ranges between canonical RFSA $\mathcal{R}(L)$ (cf. Definition 2.3.15) and minimal DFA $\mathcal{A}_L$).*

Unfortunately, Denis et al. do not give any statement on the time complexity of their DeLeTe2 algorithm. Hence, we now prove it to be polynomial in the size of the prefix tree acceptor, the alphabet and the size of the sample.

Theorem 3.2.13 (DeLeTe2: Complexity)**.** *Provided with an input sample $S = (S_+, S_-)$, the DeLeTe2 algorithm runs in time $\mathcal{O}(s \cdot k^2 + |\Sigma| \cdot k^3)$, where $s = |S|$ is the size of the input sample and $k = |Q_+|$ is the number of states of the prefix tree acceptor $\mathcal{A}(S_+)$.*

Proof: Let us consider the algorithm from Table 3.3. As we have $k = |\mathcal{A}(S_+)|$ words in *Pref* (line 2), the algorithm stops after at most k iterations. In each iteration, first maximally k state-equivalence checks have to be performed (line 6). To be precise, in total, at most $\Sigma_{1 \leq i \leq k-1} i = \frac{k \cdot (k-1)}{2}$ such checks have to be done. For each such equivalence check, the sets S_+ and S_- have to be searched in time $\mathcal{O}(|S|)$. Then either $u\Sigma^*$ is deleted from *Pref* (line 8), or transitions are added from and to the newly created state u (lines 17, 18). The first problem can be solved in $\mathcal{O}(k)$ (by exhaustively searching set *Pref*) and for the second case at most $2k \cdot |\Sigma|$ transitions are added for which, every time, a state-inclusion check ($u \prec v$) has to be performed in time $\mathcal{O}(k)$. Altogether, this yields a total time complexity of $\mathcal{O}(|S| \cdot k^2 + |\Sigma| \cdot k^3)$. □

As RPNI, the DeLeTe2 algorithm is a passive offline algorithm, which, provided with an a priori fixed set of examples, derives a possible hypothesis. Therefore, like the extension of RPNI to RPNI^2, DeLeTe2 is extendible to a passive online version (DeLeTe2^2) being provided with a stream of words it has no bearing on. This yields an iterative algorithm that can *identify in the limit* the class of regular languages using DFA as representation. An active online version for deriving canonical RFSA will be presented in Chapter 4.

3.3 Online Learning

Now let us focus on online learning algorithms (cf. Figure 3.1 on page 22). As initially stated, we subdivide this class into passive and active algorithms.

As already seen previously, a problem that often arises in the setting of (passive) offline learning is that of not providing enough information about the target language in order to derive a model with a minimal number of states. As, in this setting, the learner cannot retrieve additional information, he might get stuck, yielding a suboptimal result for certain types of applications. The RPNI and the DeLeTe2 algorithm are examples of such learning algorithms. As long as the samples do not fulfill certain properties, the resulting DFA or RFSA, respectively, can not be guaranteed to be minimal, or, in case of DeLeTe2, even to be RFSA at all. Hence, having a regular language in mind (as defined in question ($Q2$) on page 23) an identification of the unique (minimal) representative becomes impossible in the presence of incomplete samples.

In some cases, e.g., for the extension RPNI^2 of RPNI discussed previously, this problem is tackled by allowing to incrementally augment the a priori given set of sample strings by passively receiving words from a random input stream. These extensions lead to the direction of a new class of learning algorithms called *online learning algorithms*. Online inference algorithms can incrementally extend hypotheses by using classified data that becomes available over time. A famous online learning model is that of the *minimal adequate teacher*, MAT for short. It was first introduced by Angluin [Ang87a] and will be described in more detail in the following. Roughly speaking, it exhibits a *teacher* who is capable of answering certain kinds of questions, which are posed by the learning algorithm. If a hypothesis is not satisfying, counterexamples, i.e., words that are not yet correctly recognized by the hypothesis, can be used to extend and improve the hypothesis.

Let us, like before, distinguish algorithms that are able to infer DFA and NFA.

3.3.1 Learning Deterministic Automata

Online learning algorithms are characterized by the additional ability of asking *queries* to some source of information to incrementally refine the hypotheses. Some, for example, are being taught by a *teacher* which knows the target language L and is thus able to answer so-called *membership queries* to guide the learner and increase the samples "on-the-fly".

This means that for a word w in question the teacher is able to tell whether or not w is in L. Thus, in contrast to pure offline learning algorithms, online learning harness the capability of augmenting their knowledge about the language in question by requesting answers to missing information. In this way, the search space for possible candidates of minimal DFA is reduced until obtaining a unique target. Or, in other words, the search through the space of all possible automata is more directed and guided than in the case of passive offline learning where the a priori fixed set of classified words is the only available source of information. Therefore, having access to a teacher, an online algorithm should always perform better (i.e., (*i*) being more accurate because it has access to an additional information source and (*ii*) having a better time complexity) in finding the correct solution to question ($Q2$) from Subsection 3.1 (i.e., detecting the minimal DFA for a fixed regular language L) than offline algorithms like Biermann or RPNI. As shown in [Gol78], the problem of identifying the correct hypothesis automaton consistent with a given set of samples is NP-complete. Nevertheless, if the aim is not to identify a (minimal) model for a fixed regular language but to find a (minimal) model consistent with a given set of samples—which corresponds to solving question ($Q1$)—RPNI and Biermann will be favored in many applications.

Subsequently, we will introduce one of the most influencing and most cited learning algorithms, called Angluin's learning approach L*. As L* will form the base of our active online learning approaches for learning NFA in Chapter 4 and for learning communicating automata in Chapter 6, we will distinguish it from the other learning algorithms mentioned before and describe it in greater detail.

Learning Deterministic Automata Using the MAT Model

After introducing several offline learning algorithms in the previous section, it remains to introduce the key concept for the synthesis approaches we will discuss in Chapters 4 and 6. The algorithm we are referring to is called L* and is one of the most influential, most cited, implemented and extended active online learning algorithms for regular languages. It builds on the *minimal adequate teacher* (MAT) model. The MAT model originally only featured a *Learner* which tries to infer a regular language L and a *Teacher* who knows L and tries to guide the *Learner* answering questions. In this thesis, we additionally distinguish between a *Teacher* and an *Oracle*. In the MAT framework two kinds of questions can be asked by the *Learner*. A *membership query* asks whether a word w is member of a target regular language L, and an *equivalence query* demands an answer to the question if the current hypothesis recognizes the target regular language. As the first kind of question in our context is easy to answer, we call the component responsible for this task the *Teacher* and the component for equivalence queries the *Oracle* because equivalence queries are harder to answer. Note, however, that conceptually, there is no reason for differentiating between them. In [Ang87b], Angluin showed that these two kinds of questions are necessary for polynomial identifiability. If either the possibility to pose membership queries or the possibility to ask equivalence queries is omitted, polynomial inference (in the sense of inferring the minimal DFA for a fixed target language L) is not possible anymore.

Learning Regular Languages Using L*

Angluin's algorithm L* [Ang87a] learns DFA by querying for certain words whether they should be accepted or rejected by the automaton in question. Later, we generalize it

```
L* (Σ):
U := {$\varepsilon$}; V := {$\varepsilon$}; T is defined nowhere;
T-Update();
repeat
    while $(T, U, V)$ is not (closed and consistent)
        do
            if $(T, U, V)$ is not consistent then
                find $u, u' \in U, a \in \Sigma$, and $v \in V$ such that $row(u) = row(u')$ and
                                                        $row(ua)(v) \neq row(u'a)(v)$;
                $V := V \cup \{av\}$;
                T-Update();
            if $(T, U, V)$ is not closed then
                find $u \in U$ and $a \in \Sigma$ such that $row(ua) \neq row(u')$ for all $u' \in U$;
                $U := U \cup \{ua\}$;
                T-Update();
    /* $(T, U, V)$ is both closed and consistent, hence, $\mathcal{H}_{(T,U,V)}$ can be derived */
    perform equivalence test for $\mathcal{H}_{(T,U,V)}$;
    if equivalence test fails then
        get counterexample $w$;
        $U := U \cup pref(w)$;
        T-Update();
until equivalence test succeeds;
return $\mathcal{H}_{(T,U,V)}$;
```

Table 3.4: L*: active online algorithm for inferring minimal DFA

```
T-Update():
for $w \in (U \cup U\Sigma)V$ such that $T(w)$ is not defined
    $T(w) := getClassificationFromTeacher(w)$;
```

Table 3.5: Function for updating table function in L*

towards learning objects that can be *represented* by DFA. This extension will enable us in Chapter 6 to learn various classes of distributed automata.

L* learns or infers a minimal DFA for a given regular language L in an online manner. In the algorithm, the *Learner*, who initially knows nothing about L, is trying to learn the minimal DFA $\mathcal{A}_L$. To this end, it repeatedly asks queries to the *Teacher* (typically the user or a blackbox system) and the *Oracle*, who both know L. Let us formally introduce the two kinds of queries the MAT framework supports (cf. Figure 3.2):

Definition 3.3.1 (Membership- and equivalence queries)**.**

- *A* membership query *consists in asking the Teacher if a word $w \in \Sigma^*$ is in L.*
- *An* equivalence query *consists in asking the Oracle whether a* hypothesized *DFA $\mathcal{H}$ is correct, i.e., whether $L(\mathcal{H}) = L$. The Oracle answers* yes *if $\mathcal{H}$ is correct, or supplies a counterexample w, drawn from the symmetric difference of L and $L(\mathcal{H})$, i.e., from the set $(L \setminus L(\mathcal{H})) \cup (L(\mathcal{H}) \setminus L)$.*

The *Learner* maintains a prefix-closed set $U \subseteq \Sigma^*$ of words that are candidates for identifying states, and a suffix-closed set $V \subseteq \Sigma^*$ of words that are used to distinguish

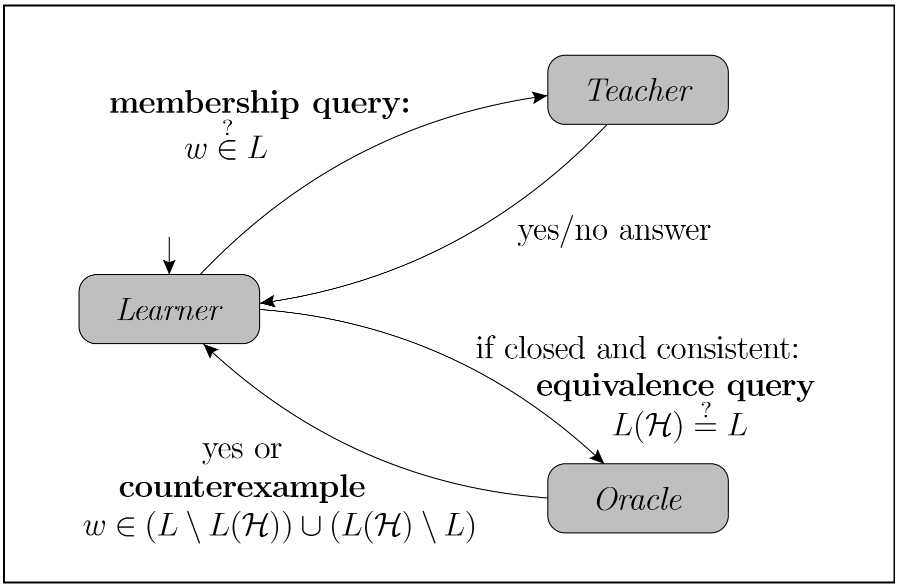

Teacher
membership query:
$w \overset{?}{\in} L$
yes/no answer
Learner
if closed and consistent:
equivalence query
$L(\mathcal{H}) \overset{?}{=} L$
yes or
counterexample
$w \in (L \setminus L(\mathcal{H})) \cup (L(\mathcal{H}) \setminus L)$
Oracle

Figure 3.2: Components of L* and their interaction

such states. The sets U and V are increased on demand. The *Learner* makes membership queries for all words in $(U \cup U\Sigma)V$, and organizes the results into a *table* $\mathcal{T} = (T, U, V)$ where function T maps each $w \in (U \cup U\Sigma)V$ to an element from $\{+, -\}$ where parity $+$ represents *accepted* and $-$ *not accepted*. To a string $u \in U \cup U\Sigma$, we assign a function $\mathit{row}(u) : V \to \{+, -\}$ given by $\mathit{row}(u)(v) = T(uv)$. Any such function is called a *row* of $\mathcal{T}$ and the set of all rows of a table is denoted by $\mathit{Rows}(\mathcal{T})$. We let $\mathit{Rows}_{\mathrm{upp}}(\mathcal{T}) = \{\mathit{row}(u) \mid u \in U\}$ denote the set of rows that represent the "upper" part of the table. Likewise, the rows from $\mathit{Rows}_{\mathrm{low}}(\mathcal{T}) = \{\mathit{row}(u) \mid u \in U\Sigma\}$ occur in its "lower" part.

The following properties of a table are relevant.

Definition 3.3.2 (Closedness and consistency). *Table* $\mathcal{T}$ *is:*

- closed *if, for all* $u \in U$ *and* $a \in \Sigma$*, there is* $u' \in U$ *such that* $\mathit{row}(ua) = \mathit{row}(u')$ *and*
- consistent *if, for all* $u, u' \in U$ *and* $a \in \Sigma$*,* $\mathit{row}(u) = \mathit{row}(u')$ *implies* $\mathit{row}(ua) = \mathit{row}(u'a)$*.*

If $\mathcal{T}$ is not closed, we find $u' \in U\Sigma$ such that $\mathit{row}(u) \neq \mathit{row}(u')$ for all $u \in U$. We move u' to U and ask membership queries for every $u'av$ where $a \in \Sigma$ and $v \in V$. Likewise, if $\mathcal{T}$ is not consistent, we find $u, u' \in U$, $a \in \Sigma$, and $v \in V$ such that $\mathit{row}(u) = \mathit{row}(u')$ and $\mathit{row}(ua)(v) \neq \mathit{row}(u'a)(v)$. Then we add av to V and ask membership queries for every $u''av$ where $u'' \in U \cup U\Sigma$. If table $\mathcal{T}$ is closed and consistent, the *Learner* constructs a hypothesized DFA $\mathcal{H}_{\mathcal{T}} = (Q, Q_0, \delta, F)$, where:

- $Q = \{\mathit{row}(u) \mid u \in U\} = \mathit{Rows}_{\mathrm{upp}}(\mathcal{T})$,
- $Q_0 = \{\mathit{row}(\varepsilon)\}$,
- δ is defined by $\delta(\mathit{row}(u), a) = \mathit{row}(ua)$ ($\mathit{row}(u) \in Q$ and $a \in \Sigma$), and
- $F = \{r \in Q \mid r(\varepsilon) = +\}$.

The *Learner* subsequently submits $\mathcal{H}_{\mathcal{T}}$ as an equivalence query to the *Oracle* asking whether $L(\mathcal{H}_{\mathcal{T}}) = L$. If the answer is affirmative, the learning procedure is completed.

Otherwise, the returned counterexample u is processed by adding every prefix of u (including u) to U, extending $U\Sigma$ accordingly, and subsequent membership queries are performed in order to make the table closed and consistent, whereupon a new hypothesized DFA is constructed, etc. (cf. Figure 3.2).

The pseudocode of L^* is given in Table 3.4, supplemented by Table 3.5, which contains the table-update function which is invoked whenever the *Teacher* is supposed to classify a word.

Theorem 3.3.3 (L^*: correctness and complexity [Ang87a])**.** *Under the assumption that the Teacher classifies/provides words in conformance with a regular language L over Σ, invoking* L^* (Σ) *eventually returns the minimal DFA $\mathcal{A}_L$. If n is the number of states of this DFA and m is the size of the largest counterexample, then the number of membership queries is in $\mathcal{O}(m \cdot |\Sigma| \cdot n^2)$ and the maximal number of equivalence queries is n. The overall running time is polynomial in m and n.*

Example 3.3.4. Assume $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w|_a = |w|_b$ and, $w = uv$ implies $|u|_b \leq |u|_a \leq |u|_b + 2\}$, i.e., for any word of L, every prefix has at least as many a's as b's and at most two more a's than b's. Moreover, the number of a's in w is equal to the number of b's. Clearly, L is a regular language over Σ. Let us illustrate how $\mathcal{A}_L$ is learned using L^*. Figure 3.3 shows several tables that are computed while learning L. The first table is initialized for $U = \{\varepsilon\}$ and $V = \{\varepsilon\}$. A table entry $T(uv)$ with $u \in U \cup U\Sigma$ and $v \in V$ has parity $+$ if $uv \in L(\mathcal{A})$ and $-$, otherwise. For example, consider Figure 3.3(i). According to the definition of L, the empty word ε is contained in L and, thus, $T(\varepsilon) = \mathit{row}(\varepsilon)(\varepsilon) = +$. In contrast, a and b are not in L, so $T(a) = T(b) = \mathit{row}(a)(\varepsilon) = \mathit{row}(b)(\varepsilon) = -$.

We deduce that table $\mathcal{T}_1$ is not closed as, e.g., $\mathit{row}(a) \notin \mathit{Rows}_{\mathrm{upp}}(\mathcal{T}_1)$. Hence, U is extended by adding a, which invokes additional membership queries. The resulting table, cf. Figure 3.3(ii) is closed and consistent, and the *Learner* presents the hypothesis automaton $\mathcal{H}_1$, which, however, does not conform to the target language L, as, e.g., $bb \in L(\mathcal{H}_1) \setminus L$. Therefore, bb and its prefix b are added to U.

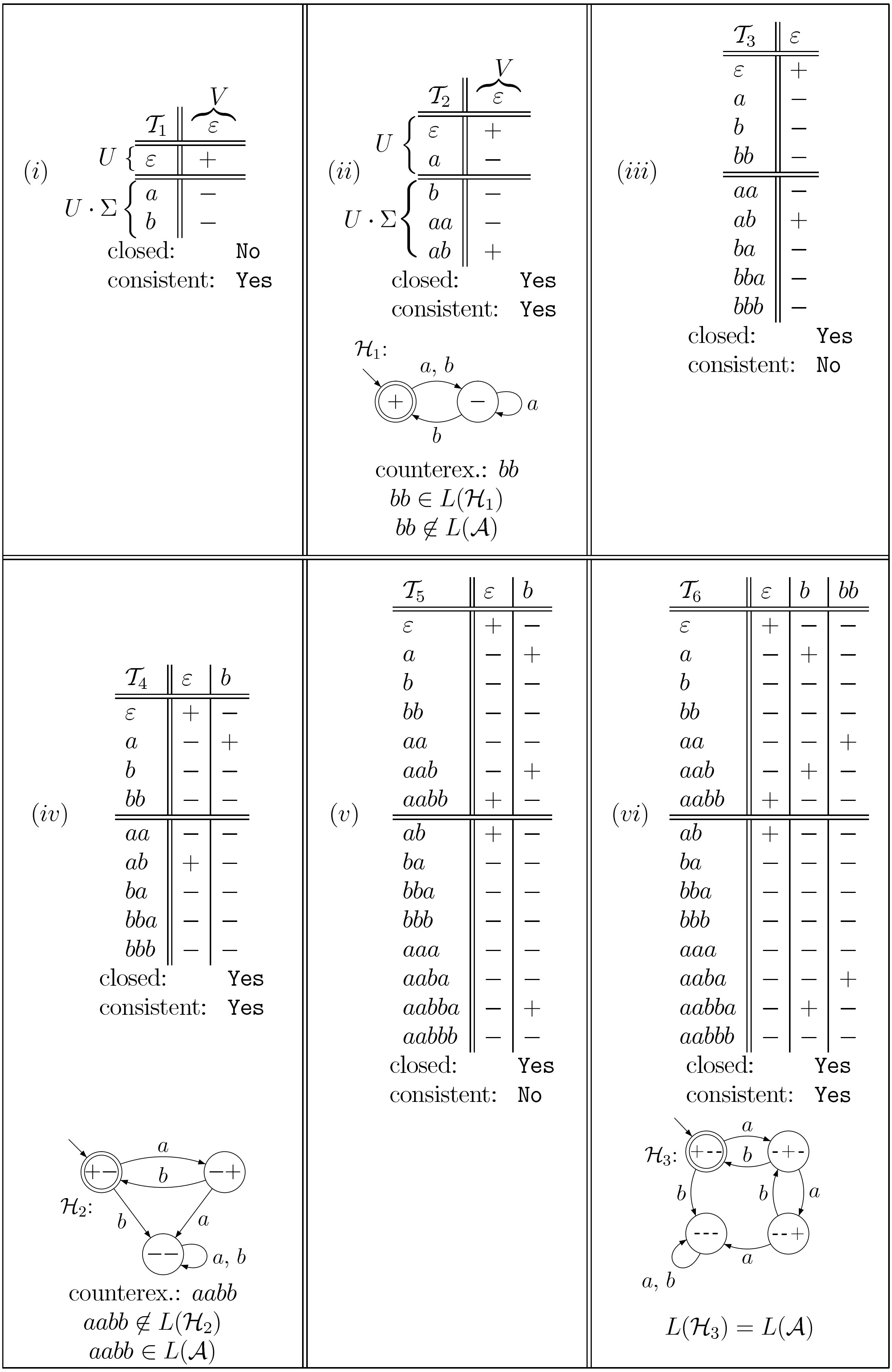
The obtained table $\mathcal{T}_3$ (Figure 3.3(iii)) is not consistent, as $\mathit{row}(a)(\varepsilon) = \mathit{row}(b)(\varepsilon) = -$ but $\mathit{row}(ab)(\varepsilon) \neq \mathit{row}(bb)(\varepsilon)$. To resolve this conflict, a column is added to the table, i.e., the set of columns V is augmented with b where b was the conflicting suffix.

Some steps later, the algorithm comes up with $\mathcal{H}_3$ (cf. Figure 3.3(vi)), which indeed recognizes L, i.e., $L(\mathcal{H}_3) = L$, so that the learning procedure finally halts. ◇

A Similar Approach: $\mathrm{L}^*_{\mathrm{col}}$

In this section, we briefly describe another version of L^* which behaves slightly different and was first introduced by [MP95] in the context of learning a subclass of ω-regular languages. We will call it $\mathrm{L}^*_{\mathrm{col}}$ as the main difference wrt. L^* is that, instead of adding counterexamples and their prefixes to the set of rows U of Angluin's table $\mathcal{T}$, we will add the counterexamples and their suffixes to the set of columns V of $\mathcal{T}$. In this way, the table $\mathcal{T}$ *always* remains consistent because there are never two identical rows in the upper table. This is because adding a counterexample and all its suffixes to V always increases the set of different rows by at least one. If this were not the case, the strings added to V would not correspond to a counterexample for the current hypothesis.

Theorem 3.3.5 ($\mathrm{L}^*_{\mathrm{col}}$: Complexity and Correctness)**.** *Under the assumption that Teacher classifies/provides words in conformance with a regular language L over Σ, invoking*



(*i*)

	$\mathcal{T}_1$	$\overbrace{\varepsilon}^{V}$
U	ε	+
$U \cdot \Sigma$	a	−
	b	−

closed: `No`
consistent: `Yes`

(*ii*)

	$\mathcal{T}_2$	$\overbrace{\varepsilon}^{V}$
U	ε	+
	a	−
$U \cdot \Sigma$	b	−
	aa	−
	ab	+

closed: `Yes`
consistent: `Yes`

counterex.: bb
$bb \in L(\mathcal{H}_1)$
$bb \notin L(\mathcal{A})$

(*iii*)

$\mathcal{T}_3$	ε
ε	+
a	−
b	−
bb	−
aa	−
ab	+
ba	−
bba	−
bbb	−

closed: `Yes`
consistent: `No`

(*iv*)

$\mathcal{T}_4$	ε	b
ε	+	−
a	−	+
b	−	−
bb	−	−
aa	−	−
ab	+	−
ba	−	−
bba	−	−
bbb	−	−

closed: `Yes`
consistent: `Yes`

counterex.: $aabb$
$aabb \notin L(\mathcal{H}_2)$
$aabb \in L(\mathcal{A})$

(*v*)

$\mathcal{T}_5$	ε	b
ε	+	−
a	−	+
b	−	−
bb	−	−
aa	−	−
aab	−	+
$aabb$	+	−
ab	+	−
ba	−	−
bba	−	−
bbb	−	−
aaa	−	−
$aaba$	−	−
$aabba$	−	+
$aabbb$	−	−

closed: `Yes`
consistent: `No`

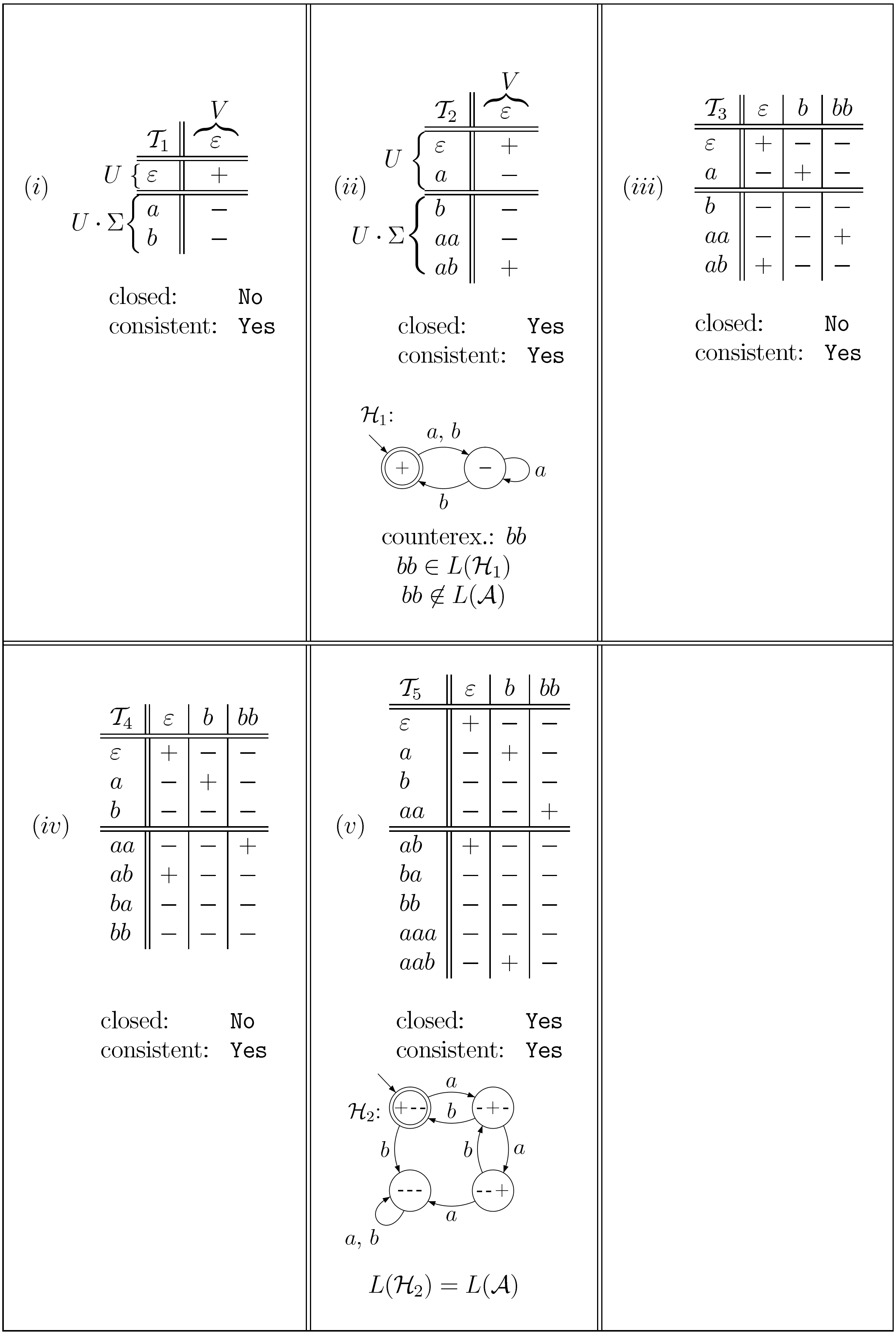
(*vi*)

$\mathcal{T}_6$	ε	b	bb
ε	+	−	−
a	−	+	−
b	−	−	−
bb	−	−	−
aa	−	−	+
aab	−	+	−
$aabb$	+	−	−
ab	+	−	−
ba	−	−	−
bba	−	−	−
bbb	−	−	−
aaa	−	−	−
$aaba$	−	−	+
$aabba$	−	+	−
$aabbb$	−	−	−

closed: `Yes`
consistent: `Yes`

$L(\mathcal{H}_3) = L(\mathcal{A})$

Figure 3.3: An example of an L* run



(i)

	$\mathcal{T}_1$	$\overbrace{\varepsilon}^{V}$
U	ε	$+$
$U \cdot \Sigma$	a	$-$
	b	$-$

closed: `No`
consistent: `Yes`

(ii)

	$\mathcal{T}_2$	$\overbrace{\varepsilon}^{V}$
U	ε	$+$
	a	$-$
$U \cdot \Sigma$	b	$-$
	aa	$-$
	ab	$+$

closed: `Yes`
consistent: `Yes`

counterex.: bb
$bb \in L(\mathcal{H}_1)$
$bb \notin L(\mathcal{A})$

(iii)

$\mathcal{T}_3$	ε	b	bb
ε	$+$	$-$	$-$
a	$-$	$+$	$-$
b	$-$	$-$	$-$
aa	$-$	$-$	$+$
ab	$+$	$-$	$-$

closed: `No`
consistent: `Yes`

(iv)

$\mathcal{T}_4$	ε	b	bb
ε	$+$	$-$	$-$
a	$-$	$+$	$-$
b	$-$	$-$	$-$
aa	$-$	$-$	$+$
ab	$+$	$-$	$-$
ba	$-$	$-$	$-$
bb	$-$	$-$	$-$

closed: `No`
consistent: `Yes`

(v)

$\mathcal{T}_5$	ε	b	bb
ε	$+$	$-$	$-$
a	$-$	$+$	$-$
b	$-$	$-$	$-$
aa	$-$	$-$	$+$
ab	$+$	$-$	$-$
ba	$-$	$-$	$-$
bb	$-$	$-$	$-$
aaa	$-$	$-$	$-$
aab	$-$	$+$	$-$

closed: `Yes`
consistent: `Yes`

$L(\mathcal{H}_2) = L(\mathcal{A})$

Figure 3.4: An example of an $\mathrm{L}^*_{\mathrm{col}}$ run

```
$L^*_{col}$ ($\Sigma$):
 1 $U := \{\varepsilon\}$; $V := \{\varepsilon\}$; $T$ is defined nowhere;
 2 T-UPDATE();
 3 repeat
 4       while $(T, U, V)$ is not closed
 5          do
 6             find $u \in U$ and $a \in \Sigma$ such that $row(ua) \neq row(u')$ for all $u' \in U$;
 7             $U := U \cup \{ua\}$;
 8             T-UPDATE();
 9       /* (T,U,V) is closed, hence, $\mathcal{H}_{(T,U,V)}$ can be derived */
10       perform equivalence test for $\mathcal{H}_{(T,U,V)}$;
11       if equivalence test fails then
12          get counterexample $w$;
13          $V := V \cup suff(w)$;
14          T-UPDATE();
15   until equivalence test succeeds;
16   return $\mathcal{H}_{(T,U,V)}$;
```

Table 3.6: L^*_{col}: a variant of Angluin's algorithm L^*

```
T-UPDATE():
 1 for $w \in (U \cup U\Sigma)V$ such that $T(w)$ is not defined
 2     $T(w) :=$ getClassificationFromTeacher($w$);
```

Table 3.7: Function for updating table function in L^*_{col}

L^*_{col} (Σ) *eventually returns the minimal DFA* $\mathcal{A}_L$. *If* n *is the number of states of this DFA and* m *is the size of the largest counterexample, then the number of membership queries is in* $\mathcal{O}(m \cdot |\Sigma| \cdot n^2)$ *and the maximal number of equivalence queries is* n. *The overall running time is polynomial in* m *and* n.

Proof: As in L^*_{col}, rows are only added when the underlying table is not closed, U can have at most n rows because there are n states in L. Thus, $U\Sigma$ contains $n \cdot |\Sigma|$ elements. When counterexamples are encountered, all their suffixes are added to V. This may happen at most n times because every counterexample increases the hypothesis size by at least one. Each counterexample has at most the length m of the longest counterexample. Hence, at most $n \cdot m$ columns are added, yielding an overall number of membership queries of $\mathcal{O}(m \cdot |\Sigma| \cdot n^2)$. Moreover, learning $\mathcal{A}_L$ may at most involve n equivalence queries, as, along the lines of L^*, every equivalence query increases the number of states by at least one. □

A pseudocode implementation of L^*_{col} is given in Tables 3.6 and 3.7. The only differences to the L^* algorithm is that the consistency check (cf. Table 3.4 lines 6–10) becomes superfluous and that instead of line 19 in Table 3.4 we now have line 13 in Table 3.6 where all suffixes of w are added to the set of columns V. A sample run of L^*_{col}, corresponding to the example presented for L^* in Example 3.3.4 and Figure 3.3, is shown in Figure 3.4. As we can see, the number of equivalence queries decreased compared to the equivalent run of L^*. In this example, even the number of membership queries dropped in comparison to Figure 3.3. This, however, is not generally the case. Usually, L^*_{col} will perform worse

```
LA($\Sigma$):
 1 Initialize $Q = \{p_0\}, Q_0 = \{p_0\}$ (where $p_0 = \varepsilon$), $\delta = \emptyset$, and $F = \emptyset$;
 2 repeat
 3         perform equivalence test for $\mathcal{H} = (Q, Q_0, \delta, F)$;
 4         if equivalence test fails
 5           then get counterexample $w$;
 6                 if $w$ is positive
 7                   then $Q = Q \cup Q(w)$ where $Q(w) = \{x \mid w = xy \in L$ for $y \in \Sigma^*\}$;
 8                        $\delta_{\text{new}} = \{p \xrightarrow{a} q \mid p, q \in Q \cup Q(w), a \in \Sigma,$ at least one of $p, q$ in $Q(w)\}$;
 9                        $\delta = \delta \cup \delta_{\text{new}}$;
10                        $F = F \cup \{w\}$;
11                   else simulate word $w$ on conjectured hypothesis $\mathcal{H}$
12                        to obtain an incorrect transition $t_{\text{bad}}$;
13                        $\delta = \delta \setminus \{t_{\text{bad}}\}$;
14 until equivalence test succeeds;
15 return $\mathcal{H} = (Q, Q_0, \delta, F)$;
```

Table 3.8: LA: active online learning algorithm for inferring RFSA

than L* considering the number of membership queries but better regarding the number of equivalence queries. This is due to the fact that in L^*_{col} much more columns are added to the table, thus distinguishing more and more rows. Therefore, a rule of thumb is to prefer L^*_{col} over L* whenever asking equivalence queries is expensive. For some statistical results concerning these statements we refer to Section 4.7 on page 66.

The reason for introducing this altered version of L* is that it resembles our new learning procedure for inferring NFA, which will be explained in the next chapter, to a larger extent and hence, simplifies comparisons between the algorithms.

3.3.2 Learning Nondeterministic Automata

This section presents an active online algorithm for deriving NFA.

LA: Learning RFSA

A first attempt to study the inference of NFA using the MAT learning model was performed by Yokomori [Yok95]. The basic idea behind his LA algorithm is described in two steps. Positive counterexamples are used to introduce new states and transitions, and negative counterexamples to delete incorrect transitions thereof. The pseudocode of the LA algorithm is given in Table 3.8.

Though this procedure is capable of inferring NFA in terms of RFSA [DLT04], the severe drawback is that given a counterexample c, the algorithm LA introduces for every prefix of c a new state (line 7). Hence, for a counterexample with a large number of prefixes, the resulting automaton can grow substantially. Therefore, this modus operandi does of course in general not yield an NFA which is smaller than the corresponding minimal DFA, let alone a unique representative.

Theorem 3.3.6 (LA: correctness and termination [Yok95]). *Given a regular language L, the* LA *algorithm derives after finitely many steps a hypothesis RFSA $\mathcal{H}$ with $L(\mathcal{H}) = L$ when provided with the correct answers to membership and equivalence queries.*

Theorem 3.3.7 (LA: complexity [Yok95]). *The time complexity of the* LA *algorithm is in* $\mathcal{O}(m^6 \cdot |\Sigma|^2 \cdot n^4)$*, where* n *is the number of states of the minimal DFA* $\mathcal{A}_L$ *and* m *is the maximal length of the counterexamples provided by* LA*. Note, however, that the size of automaton* $\mathcal{H}$ *is (at best) polynomial in the length of the longest counterexample.*

Thus, Yokomori derived the first polynomial time online learning algorithm for RFSA but is, in general, neither able to infer the canonical RFSA of a regular language L, nor to infer RFSA that are at most the size of $\mathcal{A}_L$.

Since Yokomori's LA approach, we are not aware of any successful attempts to tackle the problem of inferring NFA by active online learning. The problem seems thus to be a challenge and worth to be tackled as it may lead to interesting applications in different fields, like pattern recognition, computational linguistics and biology, or formal verification. Chapter 4 introduces an approach to this problem.

3.4 Advantages and Disadvantages

Advantages and disadvantages of the prementioned learning approaches very much depend on the field of application. Thus, we will restrict ourselves in this section to list the pros and cons wrt. our settings. In Chapter 6 and Part III we will introduce synthesis approaches that build on learning algorithms in the setting of computer-user-interaction, where the user has a certain regular language in mind and plays the role of a *Teacher* answering queries about this language which are asked by the learning algorithm. Moreover, as a second necessity, we want *Learners* that are able to derive minimal models. To solve the first requirement, we search for inference algorithms that allow for interactively classifying certain input. As passive offline algorithms do not provide such means, for our setup we have to restrict to online learning algorithms. Therefore, in our approaches offline algorithms like RPNI, Biermann or DeLeTe2 play a minor role, as they need an a priori fixed set of classified words. As secondly, we are interested in minimal models, we would have to ensure completeness or, using Biermann, solve the satisfiability problem for Boolean formulas which is known to be NP complete. Thus we might—for certain hard learning instances—run into trouble when trying to infer a smallest minimal target model.

One might be tempted to think that the class of passive online algorithms tends to be better suited for our needs because incremental extension of the input data is supported. But this conclusion is unfortunately wrong. As the *Teacher* (i.e., the user) has a regular language in mind (cf. question ($Q2$) on page 23), even if implemented in an iterated manner yielding a passive online algorithm, the learners from Section 3.2 would have an exponential worst case time complexity. Intuitively, this is because passive algorithms enumerate the hypothesis automata. With each additional query that is answered a new hypothesis might arise. Several of these hypotheses might have the same size. In the worst case, given a regular language L and a minimal DFA $\mathcal{A}_L$ recognizing L, the offline learning algorithm would have to output exponentially (in the size n of $\mathcal{A}_L$) many intermediate hypotheses until reaching the final target automaton $\mathcal{A}_L$. Angluin's algorithm L* circumvents this problem by introducing two new kinds of queries called membership and equivalence queries. With these types of queries, the algorithm obtains answers to questions of the forms "*Is a certain word in the language we want to infer*" and "*Does the currently presented hypothesis recognize the target language*", and thereby assures a maximal number of $n-1$ intermediate hypotheses before arriving at the correct target automaton (assuming that the *Teacher* classifies the words according to L). As long as the learning algorithm is not able to ask both kinds of questions, it will not be efficiently usable in the context of learning discussed in Chapters 6, 7, and 9.

Hence, neither passive offline nor passive online algorithms are the final solution to our synthesis problems. And neither is the NFA active online learning approach by Yokomori. Though, the target automata induced by his algorithm `LA` are RFSA, they are in general not canonical and might sometimes even be larger than the minimal DFA that is derivable using Angluin's L* approach. Thus, in the following, we are going to concentrate on inference algorithms that are valuable for our synthesis approaches, namely *active online* algorithms. Nevertheless, most offline and online algorithms regarded in this and the following chapters are implemented in our learning framework `libalf`, which we will report on in Chapter 8.

3.5 Summary

Combining the results from Theorem 3.1.3 from Subsection 3.1.1 and Theorem 3.2.11 from Subsection 3.2.2, we obtain the following results concerning identification in the limit:

Corollary 3.5.1 ([Gol78, dlH97, DLT04]).

- *The representation class of DFA over Σ is identifiable in the limit from polynomial time and data using the class of DFA as representation.*
- *The representation class of NFA over Σ is not identifiable in the limit from polynomial time and data using the class of NFA as representation.*
- *The representation class of NFA over Σ is not identifiable in the limit from polynomial time and data using the class of RFSA as representation (unless $|\Sigma| = 1$).*
- *The representation class of RFSA over Σ is identifiable in the limit from polynomial time and data using the class of DFA as representation.*

Table 3.9 sums up the most interesting characteristics of the learning algorithms regarded in this (and the next) chapter. It provides information about (i) the algorithms' complexities, (ii) whether the algorithms apply to ($Q1$) or ($Q2$) from page 23, and (iii) lists their main drawbacks and advantages.

Variables in the complexity considerations of Table 3.9 have to be understood as follows:

- k: size of prefix tree acceptor $\mathcal{A}(S_+)$ on input sample $S = (S_+, S_-)$,
- l: sum of lengths of elements from S_- for sample $S = (S_+, S_-)$ ($l = \Sigma_{w \in S_-} |w|$),
- s: size of the input sample S,
- n: size of minimal DFA $\mathcal{A}_L$,
- $|\Sigma|$: size of the alphabet of $\mathcal{A}_L$, and
- m: length of the longest counterexample.

The rest of Part I is organized as follows: in the next chapter will establish an online learning algorithm for deriving canonical RFSA which builds on Angluin's L* algorithm. Chapter 5 introduces an improvement which can in general be applied to Angluin-style (i.e., table-based) learning algorithms, and which will be important in the setting of learning distributed systems discussed in Chapter 6 as it significantly reduces memory consumption.

Algorithm	**Complexity**	**(Q1)**	**(Q2)**	**Drawbacks**	**Advantages**
Biermann	$\mathcal{O}(\log(k)2^k)$	×		• SAT solving approach may be slow	• though passive, the Biermann algorithm may derive a smallest minimal DFA consistent with the input
RPNI	$\mathcal{O}(l\|\Sigma\|k^4)$	×		• if input is not complete: – hypothesis will not be a minimal DFA	• even if input is not complete: – hypothesis will be consistent with input
DeLeTe2	$\mathcal{O}(sk^2 + \|\Sigma\|k^3)$	×		• if input not complete: – no guarantee that hypothesis is consistent with input, – no guarantee that hypothesis is an RFSA, • if input is complete: – no guarantee that hypothesis is a canonical RFSA	• if input is complete: – an RFSA can be inferred whose size is between canonical RFSA and minimal DFA
L*	mq: $\mathcal{O}(m\|\Sigma\|n^2)$ eq: $\mathcal{O}(n)$		×	• DFA may be exponentially larger than RFSA or other models	• often used in applications
$\mathrm{L}^*_{\mathrm{col}}$	mq: $\mathcal{O}(m\|\Sigma\|n^2)$ eq: $\mathcal{O}(n)$		×	• usually more membership queries than needed with L*	• usually less equivalence queries than needed with L*
LA	mq: $\mathcal{O}(m^6\|\Sigma\|^2n^4)$ eq: $\mathcal{O}(n)$		×	• no guarantee that resulting automaton is at most the size of minimal DFA	• RFSA can be inferred in an active, online fashion
NL* (presented in Chapter 4)	mq: $\mathcal{O}(m\|\Sigma\|n^3)$ eq: $\mathcal{O}(n^2)$		×	• slightly worse theoretical complexity results than for L*, $\mathrm{L}^*_{\mathrm{col}}$	• a learning algorithm capable of inferring canonical RFSA, • usually faster in practice than L*, $\mathrm{L}^*_{\mathrm{col}}$, • exponentially more succinct hypotheses than L*, $\mathrm{L}^*_{\mathrm{col}}$

Table 3.9: An overview over the characteristics of the presented learning algorithms

4 Learning Nondeterministic Automata

In this chapter we explain how to extend the previously introduced algorithm L* for inferring DFA, towards the learning of NFA in terms of *residual finite-state automata* (RFSA) as defined in Section 2.3.2 on page 16. As we will see, this task is not straightforward and involves some thorough and profound considerations.

Beginning with a naïve approach for implementing an online learning algorithm for RFSA, for which we show severe drawbacks, we subsequently provide a correct version thereof, which—following Angluin's L* algorithm—we will call NL*. We describe how to derive RFSA from given tables, prove the correctness of our approach, and give an extended example showing NL* in use. Next, we consider two more reasonable ideas for realizing an online learning algorithm for RFSA but show that they suffer from some severe problems like inefficiency or, in certain cases, even non-termination. Afterwards, we point out how our algorithm NL* can be adapted to cope with universal automata and highlight situations in which the new algorithm could be of use. Experiments show the practicability of the new active online learning algorithms and reveal the benefits over the L* algorithm. After collecting some lessons learned about NL*, we close this chapter by giving a brief glimpse beyond the learning of nondeterministic- and universal automata.

As described in the previous chapter, learning techniques have become increasingly prominent in a variety of areas in computer science. Especially in the domain of automatic verification great efforts have been made to improve existing techniques employing learning methods. Examples are *minimizing* (partially) specified systems [OS01], model checking *black-box systems* [PVY02], and *regular model checking* [BJNT00, AJNS04, HV05, VSVA04, VSVA05]. Almost all algorithms available learn deterministic finite-state automata, as the class of DFA has pleasant properties in the setting of learning. One important feature of DFA is that for every regular language L there is a unique minimal DFA accepting L. Thanks to Nerode's right congruence (see Definition 2.2.3), this property can be characterized, and is exploited within most available learning algorithms.

Nevertheless, the class of DFA also exhibits serious drawbacks. In many application areas, e.g., in formal verification, small automata are needed to efficiently verify certain properties of the system at hand. As, in general, a minimal DFA may be exponentially larger than an equivalent nondeterministic automaton, for many applications it would be a tremendous improvement to work with an exponentially more succinct NFA rather than with the corresponding minimal DFA. This motivates the need for learning algorithms being capable of inferring some kind of minimal NFA instead of minimal DFA. As, however, the class of NFA lacks important properties that are essential for current learning algorithms, e.g., there is no unique minimal NFA for a given regular language, it is not clear which automaton to learn. Another property that is absent in the setting of NFA is that there is no characterization of NFA in terms of right-congruence classes.

In one of their fundamental research papers, Denis et al. [DLT02] define the class of residual finite-state automata (RFSA) as introduced in Definition 2.3.13. It is a proper subset of the class of NFA but shares important properties with the class of DFA: for every regular language there is a unique canonical RFSA accepting it. Moreover, the states of this canonical RFSA correspond to right-congruence classes or, equivalently, to *residuals* (or *residual languages*) of the accepted language. Important to mention is the property that at the same time an RFSA can be exponentially more succinct than its minimal deterministic counterpart. This pleasant property turns RFSA into the preferable choice for learning regular languages. In another paper [DLT04], Denis et al. provided the passive offline learning algorithm DeLeTe2 presented in Paragraph 3.2.2, working in the spirit of RPNI. Alternatives and extensions to this algorithm have afterwards been presented, most recently in [GdPAR08], but are all based on the passive offline learning idea. So far, for active online learning RFSA, there has only been one attempt presented in [Yok95] under the name LA which infers RFSA (cf. Subsection 3.3.2), but unfortunately not the minimal canonical version thereof. In this algorithm, the derived automaton might even be larger than the corresponding minimal DFA.

Instead of being satisfied with this suboptimal online learning algorithm for NFA, we now start to collect some ideas which could emerge when trying to implement a new online algorithm for inferring NFA. As we will see, this naïve approach will not result in a satisfactory solution either, yet yields important insights leading us to the right direction for deriving a new online learning algorithm for canonical RFSA.

4.1 A Naïve Approach: Employing L^*

An initial idea for online-learning nondeterministic automata in terms of RFSA could be to employ Angluin's L^* algorithm, presented in the previous chapter. At first, we let L^* run and learn the regular language L in mind in terms of its minimal DFA. If Angluin's table $\mathcal{T}$ is fully constructed, every row in the upper part of the table corresponds to a state in the minimal DFA for L. As already seen in the preliminaries (cf. Subsection 2.3.2), states in DFA may be composed of others and, thus, be omissible in a nondeterministic version thereof. Hence, we could check, which states are composed in this way. I.e., we search for states $q_0, q_1, \ldots, q_n$ such that $L_{q_0} = L_{q_1} \cup \ldots \cup L_{q_n}$. As we will show in Section 4.2 this language inclusion (i.e., $L_{q_0} \supseteq L_{q_i}$, $1 \leq i \leq n$) can be efficiently checked by testing a syntactic property of the table rows. Nevertheless, the severe drawback of this simple approach is that it can be extremely inefficient. Assuming that the canonical RFSA in question has n states and the minimal DFA $m \geq n$ states (and hence table $\mathcal{T}$ at least m rows in the upper table), we would have to check $\mathcal{O}(m^2)$ rows for inclusion and, to obtain a consistent automaton, moreover test, if the successors of these composing states also compose the successor of the composed state. As we will show, there are canonical RFSA for which the corresponding minimal DFA is exponentially larger, yielding an exponential complexity ($\mathcal{O}((2^n)^2)$) in the number of states of the canonical RFSA.

The approach of employing L^* for deriving canonical RFSA is unsatisfactory because of two reasons: Firstly, this solution is not direct in that it uses an existing learning technique and reuses its output for calculating the canonical RFSA. A second drawback entailed by the first one is that the algorithm suffers from great inefficiency because it first computes the complete table for the minimal DFA and afterwards starts a compression yielding a canonical RFSA. As shown above, this compression is costly as a comparison of every row (of the upper table) with every other has to be performed.

The obvious conclusion is that we have to exploit inclusion relations between states as early as possible in the learning phase and, as far as possible, only create states or rows in the table, respectively, that are needed to obtain the final hypothesis automaton. Thus, we now opt for a solution employing "on-the-fly compression" during the learning phase.

4.2 NL^*: From Tables to RFSA

Having discussed an initial idea, which unfortunately did not lead to a satisfactory solution, we will now introduce the algorithm NL^* as the first and only (active online) learning algorithm capable of deriving canonical RFSA. Thus, using the MAT model featuring membership and equivalence queries, our algorithm infers a canonical RFSA for the desired language, which is always smaller than or equal to the corresponding minimal DFA.

To unify our presentation, we will closely follow Angluin's notions and notation. Like in the previous chapter, we also use tables $\mathcal{T} = (T, U, V)$ with a prefix-closed set of words $U \subseteq \Sigma^*$ representing rows, a suffix-closed set $V \subseteq \Sigma^*$ representing columns, and a mapping $T : (U \cup U\Sigma)V \to \{+, -\}$ handling the classification of words in the table. As beforehand, we associate with any word $u \in U \cup U\Sigma$ a mapping $row(u) : V \to \{+, -\}$. Again, members of U are used to reach states and members of V to distinguish states. We moreover adopt notations introduced before such as $Rows(\mathcal{T})$, $Rows_{\mathrm{upp}}(\mathcal{T})$, and $Rows_{\mathrm{low}}(\mathcal{T})$.

The main difference in the new approach is that *not all rows* of the table will correspond to states of the hypothesized RFSA, but only so-called *prime* rows. Essentially, we have to define for rows what corresponds to *union*, *composed*, *prime*, and *subset* previously introduced for languages (cf. Subsection 2.3.2).

Definition 4.2.1 (Join Operator). *Let $\mathcal{T} = (T, U, V)$ be a table. The* join $(r_1 \sqcup r_2) : V \to \{+, -\}$ *of two rows $r_1, r_2 \in Rows(\mathcal{T})$ is defined component-wise for each $v \in V$: $(r_1 \sqcup r_2)(v) := r_1(v) \sqcup r_2(v)$ where $- \sqcup - = -$ and $+ \sqcup + = + \sqcup - = - \sqcup + = +$.*

Note that the join operator is associative, commutative, and idempotent, yet that the join of two rows is *not* necessarily a row of table $\mathcal{T}$.

Definition 4.2.2 (Composed and Prime Rows). *Let $\mathcal{T} = (T, U, V)$ be a table. A row $r \in Rows(\mathcal{T})$ is called* composed *if there are rows $r_1, \ldots, r_n \in Rows(\mathcal{T}) \setminus \{r\}$ such that $r = r_1 \sqcup \ldots \sqcup r_n$. Otherwise, r is called* prime. *The set of prime rows in $\mathcal{T}$ is denoted by $Primes(\mathcal{T})$. Moreover, we let $Primes_{\mathrm{upp}}(\mathcal{T}) = Primes(\mathcal{T}) \cap Rows_{\mathrm{upp}}(\mathcal{T})$.*

Definition 4.2.3 (Covering Relation). *Let $\mathcal{T} = (T, U, V)$ be a table. A row $r \in Rows(\mathcal{T})$ is* covered *by row $r' \in Rows(\mathcal{T})$, denoted by $r \sqsubseteq r'$, if for all $v \in V$, $r(v) = +$ implies $r'(v) = +$. If moreover $r' \neq r$, then r is* strictly covered by *r', denoted by $r \sqsubset r'$.*

Note that r may be strictly covered by r' and both r and r' are prime. A composed row covers all the primes it is composed of.

Example 4.2.4. Consider, for example, the table $\mathcal{T}_3$ from Figure 4.1 which contains seven rows divided into three upper rows and four lower rows, and three columns. All rows in the upper table are primes as they cannot be subdivided into other rows of this table. In contrast, row $row(ab)$ is composed of at least two other rows. Thus, joining, e.g., $row(a)$ and $row(aa)$, we get $row(ab)$ by abuse of notation (of rows and tuples): $row(ab) = row(a) \sqcup row(aa) = (-, +, -) \sqcup (+, -, +) = ((- \sqcup +), (+ \sqcup -), (- \sqcup +)) = (+, +, +)$. The decomposition of a row into others, however, needs not be unique. Hence, another possibility for composing $row(ab)$ would also be to add $row(\varepsilon)$ to the join operation. The rows of the tables from Figure 4.1 containing a leading asterisk represent prime rows, whereas the ones without "$*$" are composed. In the following, we will always refer to lines with asterisk as prime rows and without asterisk as composed rows.

	$\mathcal{T}_1$	ε	a
$*$	ε	$-$	$-$
$*$	a	$-$	$+$
$*$	b	$-$	$-$
$*$	aa	$+$	$-$
	ab	$+$	$+$

	$\mathcal{T}_2$	ε	a
$*$	ε	$-$	$-$
$*$	a	$-$	$+$
$*$	aa	$+$	$-$
$*$	b	$-$	$-$
	ab	$+$	$+$
$*$	aaa	$-$	$+$
$*$	aab	$-$	$-$

	$\mathcal{T}_3$	ε	a	aa
$*$	ε	$-$	$-$	$+$
$*$	a	$-$	$+$	$-$
$*$	aa	$+$	$-$	$+$
$*$	b	$-$	$-$	$+$
	ab	$+$	$+$	$+$
	aaa	$-$	$+$	$+$
$*$	aab	$-$	$-$	$+$

Figure 4.1: Three example tables

Table $\mathcal{T}_3$ contains several rows covering other rows. E.g., $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(aa)$, as at every position (i.e., in this example only position aa) where $\mathit{row}(\varepsilon)$ exhibits a $+$ (i.e., $[\mathit{row}(\varepsilon)](aa) = +$) also $\mathit{row}(aa)$ features a $+$ (i.e., $[\mathit{row}(aa)](aa) = +$). As $\mathit{row}(\varepsilon) \neq \mathit{row}(aa)$ this covering relation is strict for these two rows. But considering, e.g., $\mathit{row}(\varepsilon)$ and $\mathit{row}(b)$, they are equal and hence, this covering relation is non-strict. In this case, both directions hold: $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(b)$ and $\mathit{row}(b) \sqsubseteq \mathit{row}(\varepsilon)$. Recall that a prime row can strictly cover other prime rows as it is the case for, e.g., $\mathit{row}(aa)$ and $\mathit{row}(\varepsilon)$. The rows considered so far were all comparable. This, of course, is not always the case. If we juxtapose, e.g., rows $\mathit{row}(\varepsilon)$ and $\mathit{row}(a)$, we see that they are incomparable: neither $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(a)$ nor $\mathit{row}(a) \sqsubseteq \mathit{row}(\varepsilon)$ holds. The reason is that both contain at least one position where one row has value $+$ and the other value $-$. ◇

Having exposed the key notions for this chapter, we now introduce concepts comparable to closedness and consistency of Angluin's L* algorithm and call them *RFSA-closedness* and *RFSA-consistency*, respectively.

Summarizing the last chapter's definitions: for DFA, closedness ensured that every row in the lower part of the table also occurred in the upper part. Different rows from the upper table helped building the set of states of the hypothetical automaton. For RFSA, this translates to the idea that each row of the lower part of the table is composed of prime rows from the upper part. Formally:

Definition 4.2.5 (RFSA-Closedness). *A table* $\mathcal{T} = (T, U, V)$ *is called* RFSA-closed *if, for each* $r \in \mathit{Rows}_{\mathrm{low}}(\mathcal{T})$, $r = \bigsqcup\{r' \in \mathit{Primes}_{\mathrm{upp}}(\mathcal{T}) \mid r' \sqsubseteq r\}$.

Note that a table is RFSA-closed if any prime row of the lower part is a prime row of the upper part of the table.

Example 4.2.6. Let us consider table $\mathcal{T}_1$ from Figure 4.1. It is not RFSA-closed because there exists a prime row, namely $\mathit{row}(aa)$, in the lower part of the table which is not present in the upper part, yet. Hence, it has to be moved to the upper part of the table yielding table $\mathcal{T}_2$. $\mathcal{T}_2$ is RFSA-closed.

Coming back to table $\mathcal{T}_3$ from Figure 4.1, we see that it is already RFSA-closed, as the only two lines that do not belong to the upper table, yet, are the non-prime rows: $\mathit{row}(aa)$ and $\mathit{row}(aba)$. In Angluin's L* algorithm, however, we would have to move them to the upper table to close $\mathcal{T}_3$, but in the case of RFSA, as these lines are not prime, and hence, can be composed of rows already present in the upper table (i.e., $\mathit{row}(aa) = \mathit{row}(a) \sqcup \mathit{row}(ab)$ and $\mathit{row}(aba) = \mathit{row}(\varepsilon) \sqcup \mathit{row}(a)$) there is no need for changing $\mathcal{T}_3$. ◇

Now we turn towards the second important ingredient for Angluin's L* algorithm: the notion of *consistency.* The idea of consistency in case of DFA was as follows: Assume that two words u and u' of the table have the same row. This suggests that both words lead to the same state of the DFA as they cannot be distinguished by words $v \in V$. Hence, they induce the same residuals. Then, however, ua and $u'a$ have to induce equal residuals as well, for any $a \in \Sigma$, i.e., from equivalent states in the automaton we have to reach equivalent states by taking an a-transition, no matter which $a \in \Sigma$ is taken. In other words, if there is some $a \in \Sigma$ and $v \in V$ such that $T(uav) \neq T(u'av)$, then the residuals induced by u and u' cannot be the same and must be distinguishable by the suffix av to be added to V.

For RFSA, if there are u and u' with $\mathit{row}(u) \sqsubseteq \mathit{row}(u')$, then this suggests that the residual induced by u is a subset of the residual induced by u'. If indeed so, then the same relation must hold for the successors ua and $u'a$. This is formally expressed as:

Definition 4.2.7 (RFSA-Consistency). *A table* $\mathcal{T} = (T, U, V)$ *is called* RFSA-consistent *if, for all* $u, u' \in U$ *and* $a \in \Sigma$, $\mathit{row}(u') \sqsubseteq \mathit{row}(u)$ *implies* $\mathit{row}(u'a) \sqsubseteq \mathit{row}(ua)$.

Example 4.2.8. Now consider table $\mathcal{T}_2$ in Figure 4.1. In Angluin's sense this table would be consistent as the upper table only contains pairwise distinct rows. This is not the case in our NL* setting. In table $\mathcal{T}_2$ two inclusion relations have to be checked: $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(a)$ and $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(aa)$. The second inclusion does not raise a problem. However, the first inclusion yields an inconsistency because $\mathit{row}(\varepsilon) \sqsubseteq \mathit{row}(a)$ but $\mathit{row}(a) \not\sqsubseteq \mathit{row}(aa)$. As depicted in Figure 4.1 this inconsistency can be resolved by adding a column aa to $\mathcal{T}_2$ generating table $\mathcal{T}_3$. $\mathcal{T}_3$ now is RFSA-closed and RFSA-consistent. ◇

With a table that is RFSA-closed and RFSA-consistent, we can associate an NFA. Later we will show that on termination of our learning algorithm, this NFA corresponds to a canonical RFSA.

Definition 4.2.9 (NFA of a Table). *For a table* $\mathcal{T} = (T, U, V)$ *that is RFSA-closed and RFSA-consistent, we define an NFA* $\mathcal{R}_\mathcal{T} = (Q, Q_0, \delta, F)$ *over alphabet* Σ *by:*

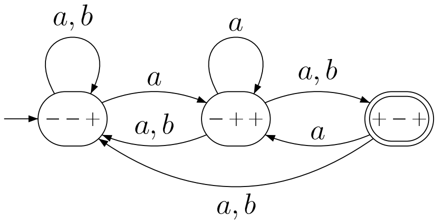
- $Q = \mathit{Primes}_{\mathrm{upp}}(\mathcal{T})$,
- $Q_0 = \{r \in Q \mid r \sqsubseteq \mathit{row}(\varepsilon)\}$,
- $\delta(\mathit{row}(u), a) = \{r \in Q \mid r \sqsubseteq \mathit{row}(ua)\}$ *for* $u \in U$ *with* $\mathit{row}(u) \in Q$ *and* $a \in \Sigma$, *and*
- $F = \{r \in Q \mid r(\varepsilon) = +\}$.

Note that $\mathit{Primes}_{\mathrm{upp}}(\mathcal{T}) = \mathit{Primes}(\mathcal{T})$, as $\mathcal{T}$ is closed. Furthermore, $\mathit{row}(\varepsilon)$ is not in Q_0 iff it is composed. Confer Figure 4.2(c) and (d) for an example with three initial states of which none exclusively represents the residual language of ε. Moreover, δ is well-defined: Take u, u' with $\mathit{row}(u) = \mathit{row}(u')$. Then, $\mathit{row}(u) \sqsubseteq \mathit{row}(u')$ and $\mathit{row}(u') \sqsubseteq \mathit{row}(u)$. Consistency implies that $\mathit{row}(ua) \sqsubseteq \mathit{row}(u'a)$ and $\mathit{row}(u'a) \sqsubseteq \mathit{row}(ua)$ so that both resulting rows are equal.

Example 4.2.10. Consider table $\mathcal{T}_1$ from Figure 4.2(a) and described in Example 4.1. We can easily convince ourselves that this table is RFSA-closed and RFSA-consistent. Hence, according to Definition 4.2.9, we can derive the NFA $\mathcal{R}_\mathcal{T} = (Q, Q_0, \delta, F)$ corresponding to $\mathcal{T}_1$:

	$\mathcal{T}_1$	ε	a	aa
$*$	ε	$-$	$-$	$+$
$*$	a	$-$	$+$	$+$
$*$	ab	$+$	$-$	$+$
$*$	b	$-$	$-$	$+$
	aa	$+$	$+$	$+$
	aba	$-$	$+$	$+$
$*$	abb	$-$	$-$	$+$

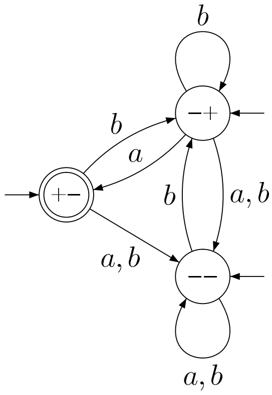
(a)



(b)

	$\mathcal{T}_2$	ε	a
	ε	$+$	$+$
$*$	a	$+$	$-$
$*$	b	$-$	$+$
$*$	aa	$-$	$-$
$*$	ab	$-$	$+$
$*$	ba	$+$	$-$
$*$	bb	$-$	$+$
$*$	aaa	$-$	$-$
$*$	aab	$-$	$+$

(c)



(d)

Figure 4.2: Two tables $\mathcal{T}_1, \mathcal{T}_2$ and their corresponding NFA $\mathcal{R}_{\mathcal{T}_1}$ and $\mathcal{R}_{\mathcal{T}_2}$

- $Q = \{row(\varepsilon), row(a), row(ab)\} = \{(-,-,+),(-,+,+),(+,-,+)\}$,
- $Q_0 = \{r \in Q \mid r \sqsubseteq row(\varepsilon)\} = \{row(\varepsilon)\}$,
- $$\begin{aligned} \delta(row(\varepsilon),a) &= \{r \in Q \mid r \sqsubseteq row(a)\} &&= \{row(\varepsilon), row(a)\},\\ \delta(row(\varepsilon),b) &= \{r \in Q \mid r \sqsubseteq row(b)\} &&= \{row(\varepsilon)\},\\ \delta(row(a),a) &= \{r \in Q \mid r \sqsubseteq row(aa)\} &&= \{row(\varepsilon), row(a), row(ab)\},\\ \delta(row(a),b) &= \{r \in Q \mid r \sqsubseteq row(ab)\} &&= \{row(\varepsilon), row(ab)\},\\ \delta(row(ab),a) &= \{r \in Q \mid r \sqsubseteq row(aba)\} &&= \{row(\varepsilon), row(a)\},\\ \delta(row(ab),b) &= \{r \in Q \mid r \sqsubseteq row(abb)\} &&= \{row(\varepsilon)\}, \text{ and} \end{aligned}$$
- $F = \{r \in Q \mid r(\varepsilon) = +\} = \{row(ab)\}$.

The automaton $\mathcal{R}_{\mathcal{T}_1}$ resulting from the application of Definition 4.2.9 to table $\mathcal{T}_1$ from Figure 4.2(a) is depicted in Figure 4.2(b). Similarly, the RFSA-closed and RFSA-consistent table $\mathcal{T}_2$ yields the NFA $\mathcal{R}_{\mathcal{T}_2}$ from Figure 4.2(d). ◇

For the rest of this section, we fix a table $\mathcal{T} = (T, U, V)$ that is RFSA-closed and RFSA-consistent. We prove some important properties of the automaton $\mathcal{R}_{\mathcal{T}}$ constructed from the table. These properties are crucial for showing that the new learning algorithm indeed infers canonical RFSA and will eventually terminate with the correct result if membership queries are answered according to a given regular language $L \subseteq \Sigma^*$.

The following lemma states that for all words $u \in U$ from the upper table the states that are reachable from Q_0 via u are covered by $row(u)$. Intuitively, this lemma ensures that we are always able to reach all states that represent prime residuals contained in the residual language of state $u^{-1}L$ of DFA $\mathcal{A}(L)$ (cf. Definition 2.3.7).

Lemma 4.2.11. *Let $\mathcal{R}_{\mathcal{T}} = (Q, Q_0, \delta, F)$. For all $u \in U$ and $r \in \delta(Q_0, u)$, we have $r \sqsubseteq row(u)$.*

Proof: We prove this lemma by induction on the length of u. If $u = \varepsilon$, then we have $\delta(Q_0, \varepsilon) = Q_0$ and by definition of $\mathcal{R}_{\mathcal{T}}$ we have $\forall r \in Q_0. r \sqsubseteq row(\varepsilon)$. If $u = u'a$, then $\delta(Q_0, u'a) = \delta(\delta(Q_0, u'), a)$. Take an $r \in \delta(\delta(Q_0, u'), a)$. Because of the definition of δ we have that there exist $u'' \in U$ and $r' \in \delta(Q_0, u')$ with $r' = row(u'')$ and $r \sqsubseteq row(u''a)$. By induction hypothesis we have $r' \sqsubseteq row(u')$. Therefore $row(u'') \sqsubseteq row(u')$ and, by RFSA-consistency, $row(u''a) \sqsubseteq row(u'a)$. This implies $r \sqsubseteq row(u'a)$. □

The next lemma says that each *state* of $\mathcal{R}_{\mathcal{T}}$ correctly classifies strings of V.

Lemma 4.2.12. *Let $\mathcal{R}_{\mathcal{T}} = (Q, Q_0, \delta, F)$. For each $r \in Q$ and $v \in V$, the following hold:*

(i) $r(v) = -$ *iff* $v \notin L_r$ *and*

(ii) $row(\varepsilon)(v) = -$ *iff* $v \notin L(\mathcal{R}_{\mathcal{T}})$.

Proof:

(i) We prove for all $v \in V.(r(v) = -$ iff $v \notin L_r)$ by induction on the length of v.

Suppose $v = \varepsilon$. Then, $r(\varepsilon) = -$ iff $r \notin F$ by definition of $\mathcal{R}_{\mathcal{T}}$ and, therefore, $r(\varepsilon) = -$ iff $\varepsilon \notin L_r$. Suppose now $v = av'$. Let $r = row(u)$ for some $r \in Q$ and $u \in U$.

"only if": As V is suffix-closed, $r(av') = row(u)(av') = -$ implies $row(ua)(v') = -$. Since the table is RFSA-closed, we have $row(ua) = \bigsqcup\{r' \in Q \mid r' \sqsubseteq row(ua)\}$. Thus, by the definition of $\sqsubseteq$, we have that for all $r' \in Q$ with $r' \sqsubseteq row(ua)$, $r'(v') = -$ as well. Due to the induction hypothesis, this implies $v' \notin L_{r'}$ for all $r' \in Q$ with $r' \sqsubseteq row(ua)$. Thus, $av' \notin L_r$, as the states reached from r by a are exactly the $r' \in Q$ with $r' \sqsubseteq row(ua)$ by definition.

"if": Now let $r(av') = row(u)(av') = +$. This implies that $row(ua)(v') = +$ as V is suffix-closed. Since the table is RFSA-closed there exists $r' \in Q$ with $r' \sqsubseteq row(ua)$ and $r'(v') = +$. Then, by induction hypothesis, $v' \in L_{r'}$. Therefore, $av' \in L_r$, since by definition of the transition function, r' can be reached from r by a.

(ii) Now, for all $v \in V.(row(\varepsilon)(v) = -$ iff $v \notin L(\mathcal{R}_{\mathcal{T}}))$ follows easily if $row(\varepsilon) \in Q$. If not, then we have $row(\varepsilon) = \bigsqcup\{r' \in Q \mid r' \sqsubseteq row(\varepsilon)\} = Q_0$, and the "only if"-direction follows from the first part of the lemma applied on r'. The "if"-direction follows from the fact that $v \notin L(\mathcal{R}_{\mathcal{T}})$ implies that for all $r' \in Q$ with $r' \sqsubseteq row(\varepsilon)$ we have $v \notin L_{r'}$, and from applying the first part of the lemma.

□

This fact will now enable us to prove a lemma stating that the covering relation precisely reflects language inclusion.

Lemma 4.2.13. *Let $\mathcal{R}_{\mathcal{T}} = (Q, Q_0, \delta, F)$. For every $r_1, r_2 \in Q$, $r_1 \sqsubseteq r_2$ iff $L_{r_1} \subseteq L_{r_2}$.*

Proof: Let $r_1, r_2 \in Q$, and assume $u_1, u_2 \in U$ with $row(u_1) = r_1$ and $row(u_2) = r_2$.

"only if": Suppose that $r_1 \sqsubseteq r_2$ and $w \in L_{r_1}$. We distinguish two cases:

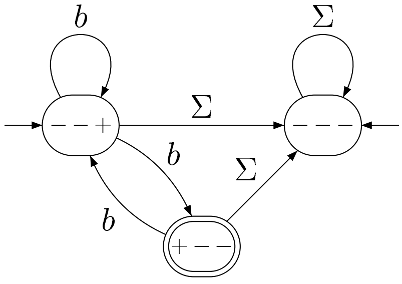
Assume first $w = \varepsilon$. Then, $row(u_1)(\varepsilon) = +$ and, due to $r_1 \sqsubseteq r_2$, $row(u_2)(\varepsilon) = +$. Thus, $r_2 \in F$ so that $\varepsilon \in L_{r_2}$. Now let $w = aw'$ with $a \in \Sigma$. We have $\delta(r_1, aw') \cap F \neq \emptyset$. Thus, there is $r \in \delta(r_1, a)$ such that $\delta(r, w') \cap F \neq \emptyset$. From $r_1 \sqsubseteq r_2$, we obtain, by RFSA-consistency, $row(u_1a) \sqsubseteq row(u_2a)$. By the definition of δ, $r \sqsubseteq row(u_1a)$, which implies $r \sqsubseteq row(u_2a)$. Thus, $r \in \delta(r_2, a)$, and we have $aw' \in L_{r_2}$.

"if": Assume $r_1 \not\sqsubseteq r_2$. We will show that $L_{r_1} \not\subseteq L_{r_2}$. By definition of the $\sqsubseteq$-relation, there has to be $v \in V$ such that $row(u_1)(v) = +$ but $row(u_2)(v) = -$. By Lemma 4.2.12, $v \in L_{r_1}$ and $v \notin L_{r_2}$. Therefore, $L_{r_1} \not\subseteq L_{r_2}$. □

Example 4.2.14. Let $L = \Sigma^* a \Sigma$ $(\Sigma = \{a, b\})$ be a regular language. It is recognized by the NFA from Figure 4.2(b). Considering the table $\mathcal{T}_1$ from Figure 4.2(a) again, we have that row $r_1 = row(a)$ strictly covers $r_2 = row(\varepsilon)$ and hence, their residual languages have to be in this inclusion relation as well. Calculating $L_{r_1} = a^{-1}L$ yields $L_{r_1} = \Sigma^* a \Sigma \cup \Sigma$

	$\mathcal{T}$	ε	ab	b
$*$	ε	$-$	$-$	$+$
	b	$+$	$-$	$+$
$*$	a	$-$	$-$	$-$
$*$	ba	$+$	$-$	$-$
	bb	$+$	$-$	$+$
$*$	ab	$-$	$-$	$-$
$*$	aa	$-$	$-$	$-$
$*$	bab	$-$	$-$	$+$
$*$	baa	$-$	$-$	$-$

(a)



(b)

Figure 4.3: A table $\mathcal{T}$ and the corresponding NFA $\mathcal{R}_{\mathcal{T}}$ which are not consistent according to Definition 4.2.15

and $L_{r_2} = \varepsilon^{-1}L = \Sigma^* a\Sigma$. Obviously $L_{r_2} \subsetneq L_{r_1}$. The same applies to r_2 and $r_3 = \mathit{row}(ab)$. Row r_2 is strictly covered by r_3 and so are their corresponding residual languages as $L_{r_3} = \Sigma^* a\Sigma \cup \{\varepsilon\}$. In contrast, r_1 and r_3 are incomparable and so are their residual languages. ◇

An important property that will distinguish Angluin's learning algorithm L* and the new learning algorithm for NFA is the following: The automaton $\mathcal{R}_{\mathcal{T}}$ constructed from the RFSA-closed and RFSA-consistent table $\mathcal{T}$ is not necessarily an RFSA (see Appendix B.3 on page 179 for an example). But we can show that $\mathcal{R}_{\mathcal{T}}$ is a canonical RFSA if it is *consistent wrt. the table* $\mathcal{T}$, i.e., the automaton correctly classifies all words of $\mathcal{T}$.

This justifies the following definition of an automaton $\mathcal{R}_{\mathcal{T}}$ consistent with a table $\mathcal{T}$.

Definition 4.2.15. *We say that automaton* $\mathcal{R}_{\mathcal{T}}$ *is* consistent with the table $\mathcal{T}$ *if, for all* $w \in (U \cup U\Sigma)V$, *we have* $T(w) = +$ *iff* $w \in L(\mathcal{R}_{\mathcal{T}})$.

If there is an automaton $\mathcal{R}_{\mathcal{T}}$ that is not consistent with its table $\mathcal{T}$, we can immediately retrieve a counterexample w (cf. Definition 4.2.15), which is passed to the learning algorithm resurrecting the learning procedure.

The next lemma is a stronger version of Lemma 4.2.11, if we have additionally that $\mathcal{R}_{\mathcal{T}}$ is consistent with $\mathcal{T}$. Under this assumption, it states that by traversing automaton $\mathcal{R}_{\mathcal{T}}$ starting in its set of initial locations with any word $u \in U$ from the upper table, the state $\mathit{row}(u) \in Q$ is reachable.

Lemma 4.2.16. *Suppose* $\mathcal{R}_{\mathcal{T}} = (Q, Q_0, \delta, F)$ *is consistent with* $\mathcal{T}$. *Then, for all* $u \in U$ *with* $\mathit{row}(u) \in Q$, *we have* $\mathit{row}(u) \in \delta(Q_0, u)$.

Proof: If $\mathit{row}(u)(v) = -$ for all $v \in V$, this is easy to see by the definition of δ. If not, we suppose that $\mathit{row}(u) \notin \delta(Q_0, u)$ and get a contradiction. With Lemma 4.2.11, we have $\forall r \in \delta(Q_0, u).r \sqsubseteq \mathit{row}(u)$. Then, Lemma 4.2.13 implies $\forall r \in \delta(Q_0, u).L_r \subseteq L_{\mathit{row}(u)}$. As $\mathit{row}(u) \in Q$ and $\mathit{row}(u) \notin \delta(Q_0, u)$, there exists $v \in V$ such that $\mathit{row}(u)(v) = +$ and for all $r \in \delta(Q_0, u)$, $r(v) = -$. This, together with Lemma 4.2.12, implies that for all $r \in \delta(Q_0, u).v \notin L_r$. But then $uv \notin L(\mathcal{R}_{\mathcal{T}})$ which is a contradiction to the fact that $\mathcal{R}_{\mathcal{T}}$ is consistent with $\mathcal{T}$. □

Example 4.2.17. In Figure 4.3(a) we see an RFSA-closed and RFSA-consistent table $\mathcal{T}$. Hence, according to Definition 4.2.9 an automaton $\mathcal{R}_{\mathcal{T}}$ can be derived (cf. Figure 4.3(b)).

On closer inspection we realize that, following table $\mathcal{T}$, word $w = ba$ should be accepted by automaton $\mathcal{R}_{\mathcal{T}}$, but this is, indeed, not the case. Consequently, table $\mathcal{T}$ and NFA $\mathcal{R}_{\mathcal{T}}$ are not consistent according to Definition 4.2.15. Moreover, as an intermediate result of the learning procedure, table $\mathcal{T}$ does not yield an RFSA, as, e.g., state $\mathit{row}(ba)$ accepts the language $L_{\mathit{row}(ba)} = bbb^*$ which is not a residual of $L(\mathcal{R}_{\mathcal{T}}) = bb^*$. ◇

We are now prepared to infer the main result of this section, stating that $\mathcal{R}_{\mathcal{T}}$ is a canonical RFSA if table $\mathcal{T}$ is RFSA-closed and RFSA-consistent, and table $\mathcal{T}$ and NFA $\mathcal{R}_{\mathcal{T}}$ are consistent according to Definition 4.2.15.

Theorem 4.2.18. *Let $\mathcal{T}$ be a table that is RFSA-closed and RFSA-consistent, and let $\mathcal{R}_{\mathcal{T}}$ be consistent with $\mathcal{T}$. Then, $\mathcal{R}_{\mathcal{T}}$ is a canonical RFSA.*

Proof: Let $\mathcal{T} = (T, U, V)$ and assume $\mathcal{R}_{\mathcal{T}} = (Q, \{q_0\}, \delta, F)$. Let L denote $L(\mathcal{R}_{\mathcal{T}})$. We first show that $\mathcal{R}_{\mathcal{T}}$ is an RFSA (i.e., all states accept residuals). Let $u \in U$ with $\mathit{row}(u) \in Q$. We show that $L_{\mathit{row}(u)} = u^{-1}L$. Due to Lemma 4.2.16 we have $\mathit{row}(u) \in \delta(Q_0, u)$. This implies $L_{\mathit{row}(u)} \subseteq u^{-1}L$. Furthermore, together with Lemma 4.2.11 we have that $\forall r \in \delta(Q_0, u).r \sqsubseteq \mathit{row}(u)$. This implies with Lemma 4.2.13 $\forall r \in \delta(Q_0, u).L_r \subseteq L_{\mathit{row}(u)}$. This gives $u^{-1}L \subseteq L_{\mathit{row}(u)}$. Together with $L_{\mathit{row}(u)} \subseteq u^{-1}L$ we have $L_{\mathit{row}(u)} = u^{-1}L$.

So far we have shown that automaton $\mathcal{R}_{\mathcal{T}}$ is an RFSA. It remains to show that it is even a canonical RFSA and, hence, that $L_{\mathit{row}(u)}$ is prime. But this is due to Lemma 4.2.13, which states that the relation $\sqsubseteq$ over rows corresponds exactly to the subset relation over languages. This precise correspondence is also the reason why the transition function δ is saturated, as required in the definition of a canonical RFSA (cf. Definition 2.3.15). □

Example 4.2.19. Consider again the regular language $L = \Sigma^* a \Sigma$ and the automaton $\mathcal{R}_{\mathcal{T}_1}$ from Figure 4.2(b). Now let us show that, indeed, the automaton $\mathcal{R}_{\mathcal{T}_1}$ derived from table $\mathcal{T}_1$ in Figure 4.2 is a canonical RFSA. Firstly, it is easy to verify that $\mathcal{T}_1$ is RFSA-closed and RFSA-consistent and that automaton $\mathcal{R}_{\mathcal{T}_1}$ is consistent with $\mathcal{T}_1$. Now it remains to check that all states correspond to prime residuals. The residual languages of states $\mathit{row}(\varepsilon)$, $\mathit{row}(a)$, and $\mathit{row}(ab)$ have already been calculated in Example 4.2.14: $\varepsilon^{-1}L = \Sigma^* a \Sigma$, $a^{-1}L = \Sigma^* a \Sigma \cup \Sigma$ and $(ab)^{-1}L = \Sigma^* a \Sigma \cup \{\varepsilon\}$. All three residuals are prime because none of them is the union of the other two, and the remaining residual of the underlying regular language is composed of the three mentioned before. This shows that $\mathcal{R}_{\mathcal{T}_1}$ really is a canonical RFSA. ◇

4.3 NL*: The Algorithm

We now describe the new learning algorithm NL* for inferring canonical RFSA. In this section we will describe the algorithm, give its pseudocode implementation, and prove NL* to be correct and efficient.

Table 4.1 describes the pseudocode implementation of NL*. The grayishly highlighted boxes specify the changes compared to the L* algorithm from Table 3.4 on page 35. As input, NL* receives the alphabet Σ it will be acting on and a target regular language $L \subseteq \Sigma^*$. Like in Angluin's algorithm L*, the aim is to infer an automaton $\mathcal{R}_{\mathcal{T}}$ fulfilling $L(\mathcal{R}_{\mathcal{T}}) = L$. In our current case the inferred automaton will be a canonical RFSA. After initializing the table $\mathcal{T}$ like in Angluin's L* algorithm (line 1), the current table is repeatedly checked for RFSA-closedness (line 5) and RFSA-consistency (line 8). If the algorithm, in lines 6 and 7, detects a violation of the RFSA-closedness condition

```
$\mathrm{NL}^*(\Sigma, L)$:
 1 initialize $\mathcal{T} := (T, U, V)$ by $U = V = \{\varepsilon\}$ and $T(w)$ for all $w \in (U \cup U\Sigma)V$
 2 repeat
 3         while $\mathcal{T}$ is not (RFSA-closed and RFSA-consistent)
 4             do
 5                 if $\mathcal{T}$ is not RFSA-closed then
 6                     find $u \in U$ and $a \in \Sigma$ such that $row(ua) \in Primes(\mathcal{T}) \setminus Primes_{\mathrm{upp}}(\mathcal{T})$;
 7                     extend table to $\mathcal{T} := (T', U \cup \{ua\}, V)$ by membership queries;
 8                 if $\mathcal{T}$ is not RFSA-consistent then
 9                     find $u \in U, a \in \Sigma$, and $v \in V$ such that the following holds:
10                         $T(uav) = -$ and
11                         $T(u'av) = +$ for some $u' \in U$ such that $row(u') \sqsubseteq row(u)$,
12                     extend table to $\mathcal{T} := (T', U, V \cup \{av\})$;
13         /* $\mathcal{T}$ is both RFSA-closed and RFSA-consistent */
14         from $\mathcal{T}$, construct the hypothesized NFA $\mathcal{R}_\mathcal{T}$ (cf. Definition 4.2.9)
15         /* perform equivalence test */
16         if $(L = L(\mathcal{R}_\mathcal{T}))$
17            then equivalence test succeeds;
18            else get counterexample $w \in (L \setminus L(\mathcal{R}_\mathcal{T})) \cup (L(\mathcal{R}_\mathcal{T}) \setminus L)$
19                 extend table to $\mathcal{T} := (T', U, V \cup suff(w))$ by membership queries;
20 until equivalence test succeeds;
21 return $\mathcal{R}_\mathcal{T}$;
```

Table 4.1: NL*: the NFA version of Angluin's algorithm L*

(cf. Definition 4.2.5), i.e., some *row*(ua) with $u \in U$ and $a \in \Sigma$ is prime and is not contained in $Primes_{\mathrm{upp}}(\mathcal{T})$, then ua is added to U. This involves additional membership queries. On the other hand, whenever the algorithm, in lines 9–11, perceives an RFSA-consistency violation (cf. Definition 4.2.7) a distinguishing suffix av can be determined which makes two existing rows distinct or incomparable. In this case, column av is added to V (line 12) invoking supplemental queries. This procedure is repeated until $\mathcal{T}$ is RFSA-closed and RFSA-consistent. If both properties are fulfilled, a conjecture $\mathcal{R}_\mathcal{T}$ can be derived (line 14) from $\mathcal{T}$ (cf. Definition 4.2.9) and be passed to the equivalence test (line 16). This test either returns a counterexample w (line 18) from the symmetric difference of $L(\mathcal{A})$ and $L(\mathcal{R}_\mathcal{T})$, and *suff*$(w)$ is added to V reinvoking NL*, or lets the learning procedure successfully terminate returning the desired automaton $\mathcal{R}_\mathcal{T}$ (lines 17, 20f.). Whenever table $\mathcal{T}$ and hypothesis $\mathcal{R}_\mathcal{T}$ are not consistent according to Definition 4.2.15 the equivalence test in line 16 will fail, yielding a counterexample w, which will revive the learning procedure. Notice that the algorithm makes sure that V is always suffix-closed and U prefix-closed, a crucial property needed throughout the proofs of this chapter.

Remark 4.3.1. We chose to treat the counterexamples as in the variant $\mathrm{L}^*_{\mathrm{col}}$ of L* described in Subsection 3.3.1. Indeed, as will be described in Subsection 4.5.2, treating the counterexamples as in the original L* does not lead to a terminating algorithm (see Appendix B.4 for an example). The treatment of counterexamples as in $\mathrm{L}^*_{\mathrm{col}}$ ensures that each row can appear at most once in the upper part of the table, because we only add rows when the table is not RFSA-closed. Note moreover that we did not explicitly include a test for consistency of the hypothesis with the table because this check is implicitly performed by the equivalence test in line 16 of Table 4.1. ◇

Proving the termination of Angluin's learning algorithm L* is quite straightforward as a simple observation can be used that directly assures termination. This key observation is that each failing equivalence query increases the overall number of states by at least one. Hence, if we would like to infer a regular language represented by a minimal DFA with n states, we would have to ask at most n equivalence queries to ensure termination. In our setting, however, this is not that easy anymore since, as we show in Appendices B.5 and B.6 on pages 183 and 185, respectively, after an equivalence query or a violation of RFSA-consistency the number of states of the hypothesized automaton does not necessarily increase and sometimes even decreases. Thus, we have to accomplish a much more sophisticated analysis of the algorithm. To show termination of NL*, we first need a simple lemma.

Lemma 4.3.2. *If the minimal DFA $\mathcal{A}_L$ for a given regular language L has n states, then the tables constructed in the runs of NL* with input $L(\mathcal{A}_L)$ cannot have more than n different rows.*

Proof: Having more than n different rows in a table implies that L has more than n different residuals, which is impossible, as the minimal DFA $\mathcal{A}_L$ for L has n residuals and hence, n states. □

Now we are prepared to derive the main contribution of this section. For any given regular language L, the algorithm NL* infers the canonical RFSA $\mathcal{R}$ accepting L.

Theorem 4.3.3. *Let n be the number of states of the minimal DFA $\mathcal{A}_L$ for a given regular language $L \subseteq \Sigma^*$. Let m be the length of the longest counterexample returned by the equivalence test (or 1 if the equivalence test always succeeds). Then,* NL* *returns after at most $\mathcal{O}(n^2)$ equivalence queries and $\mathcal{O}(m|\Sigma|n^3)$ membership queries the canonical RFSA $\mathcal{R}(L)$.*

Proof: First of all, if the algorithm terminates, then it outputs the canonical RFSA for L due to Theorem 4.2.18, because passing the equivalence test implies that the constructed automaton must be consistent with the table.

We show in the following that the algorithm terminates after at most $\mathcal{O}(n^2)$ equivalence queries. We first define a measure $\mathfrak{M}$ associating a tuple of positive natural numbers to tables. For a given table $\mathcal{T}$, let $\mathfrak{M}(\mathcal{T}) = (l_{up}, l, p, i)$, where $l_{up} = |\mathit{Rows}_{\mathrm{upp}}(\mathcal{T})|$ is the number of rows in the upper part of the table, $l = |\mathit{Rows}(\mathcal{T})|$ the number of different rows in the whole table, $p = |\mathit{Primes}(\mathcal{T})|$ the number of prime rows in the table, and i the number of strict coverings of pairs of different rows of the table, i.e., $i = |\{(r, r') \mid r, r' \in \mathit{Rows}(\mathcal{T}) \text{ and } r \sqsubset r'\}|$. It is crucial to consider rows and not members of U. Initially, $l_{up} = 1$ and ($l = 1$ or $l = 2$) and ($p = 1$ or $p = 2$) and ($i = 0$ or $i = 1$).

Let us examine how the measure (l_{up}, l, p, i) evolves during a run of NL*. A detailed example of this evolution is given after this proof in Example 4.3.4. It is clear that l_{up} and l can never decrease since two different rows stay different by extending the table.

Now let us consider the three possible cases during an NL* run which have an influence on $\mathfrak{M}(\mathcal{T})$, namely: $\mathcal{T}$ is (i) not RFSA-closed, (ii) not RFSA-consistent, and (iii) RFSA-closed and RFSA-consistent but there is a counterexample.

(i) If the table is not RFSA-closed, then, after extending the table, l_{up} increases by one. Simultaneously, l might increase by $0 \leq k \leq |\Sigma|$ (the number of new rows added). At the same time, i might increase by at most k times the old value of l (the largest possible number of strict covering relations between new rows and old rows) plus

$k(k-1)/2$ (the largest possible number of strict covering relations between new rows).

(ii) If the table is not RFSA-consistent, then, after extending the table, l_{up} stays unchanged. However, l might increase by $0 < k' \leq n$. At the same time, as before, i might increase by at most k' times the old value of l plus $k'(k'-1)/2$. If l does not increase, then this means that, for any two strings $u, u' \in U \cup U\Sigma$ with $row(u) = row(u')$ in the table before the extension, we have again $row(u) = row(u')$ after the extension. Therefore, no strict covering relation can be added in the extended table. But since we add a word to V making two rows r and r' in the original table with $r \sqsubset r'$ incomparable, i is *decreased* by at least one.

(iii) If the table is RFSA-closed and RFSA-consistent, then an equivalence query is performed. Let us fix an RFSA-closed and RFSA-consistent table $\mathcal{T} = (T, U, V)$ before the equivalence test. If the test fails, we obtain a counterexample w and a new table $\mathcal{T}' = (T', U, V \cup \mathit{suff}(w))$. Notice that $\mathcal{T}$ must be extended (otherwise, we have $w \in V$, which implies with Lemma 4.2.12 that w is correctly classified by $\mathcal{R}_{\mathcal{T}}$). Either l increases or not.

- If l increases by $0 < k'' \leq n$, then, as before, i might increase by at most k'' times the old value of l plus $k''(k''-1)/2$.
- If l does not increase, then i cannot increase (see explanation for the case that the table is not RFSA-consistent). We will furthermore show that p increases or i decreases. Suppose that this is not the case, i.e., p and i remain unchanged[1]. Then, the automata $\mathcal{R}_{\mathcal{T}}$ and $\mathcal{R}_{\mathcal{T}'}$ constructed from $\mathcal{T}$ and, respectively, $\mathcal{T}'$ must be the same: all primes of $\mathcal{T}$ must still be primes in $\mathcal{T}'$ (as p stays the same, no primes are added), the initial and final states stay the same, and the transition function is defined using the covering relation which does not change. This is because l does not change and, therefore, no new strict covering relation can be added like for the corresponding case above, where the table is not RFSA-consistent. Furthermore, since i does not change, no strict covering relation is removed. But the two automata being the same is a contradiction since $\mathcal{R}_{\mathcal{T}'}$ classifies w correctly according to Lemma 4.2.12 (w is in V), whereas $\mathcal{R}_{\mathcal{T}}$ does not. Therefore, p increases or i decreases (notice that p might be decreased by other steps).

Putting the three different cases together, we notice that after each extension of the table either:

(1) l_{up} is increased by one, or (case (i))

(2) l is increased by some bounded value $0 < k \leq n$ and simultaneously i is increased by at most $kl + k(k-1)/2$, or (cases (i)-(iii))

(3) l stays the same and we have that i decreases or p increases. (case (iii))

Moreover, due to Lemma 4.3.2, l_{up}, l, and p cannot increase beyond n which means:

(A) l_{up} can increase at most n times.

---

[1] Note that p cannot decrease if l stays constant since that would mean that now a row turns out to be composable by other rows, which is a contradiction since no new row was added.

(B) l can increase at most n times.

  - Every time l increases by $k > 0$, i increases at most by $kl + k(k-1)/2$, where the sum of all k with which l increases can be at most n. This in turn means, that i can at most increase by $\mathcal{O}(n^2)$ in one complete NL* run.

(C) If l stays unchanged, then

  1) p stays unchanged and i decreases, or
  2) p increases and i stays unchanged.

Summarizing, there are four possibilities: (A), (B), (C.1), and (C.2). Intuitively the algorithm performs sequences of the form (A), (B), (C.1), (C.2), (C.1), (B), etc., the complete sequence can contain at most n (A) elements. If you abstract from the (A)s in the sequence, there are at most n subsequences where only (C.1) and (C.2) occur (i.e., without interleaving (B)). This is because l can only be increased up to n and as l never decreases there can in total only be n occurrences of (B). Considering the (C.1) and (C.2) sequences (i.e., without interleaving (B)s) we see that they can at most be of length $n+i$. This is because p can be increased at most n times and i can be decreased at most to 0.

In total, we have at most n times (A), n times (B), $\mathcal{O}(n^2)$ times (C.1) (because, in total, i will be increased at most $\mathcal{O}(n^2)$ times and is always greater or equal to 0, thus it cannot be decreased more than $\mathcal{O}(n^2)$ times), and $n \cdot n$ times (C.2). As after each equivalence query we have (A), (B), or (C.1), or (C.2) we can have at most $\mathcal{O}(n^2)$ equivalence queries.

Hence, the algorithm must (1) always reach an equivalence query and (2) terminate after at most $\mathcal{O}(n^2)$ equivalence queries.

Concerning the number of membership queries, we notice that their maximal number corresponds to the size of the table which has at most $n + n|\Sigma|$ (n rows in the upper part + their successors) rows and $\mathcal{O}(mn^2)$ columns since at each extension at most m suffixes are added to V, yielding an overall membership query complexity of $\mathcal{O}(m|\Sigma|n^3)$. □

The theoretical complexity we obtain for NL* in terms of equivalence queries is higher compared to L* where at most n equivalence queries are needed. The complexity in terms of membership queries is higher for NL* as well (L* needs roughly $m|\Sigma|n^2$ queries). But, as we will observe in Section 4.7, in our experiments *far less* equivalence and membership queries are needed. An interesting case is presented in Appendix B.7. There, NL* needs 6 equivalence queries for a minimal DFA with only 5 states. As this cannot happen using L* this example might be illuminating for finding hard instances for NL* in terms of equivalence queries.

To gain a better intuition for the previous proof, we visualize the modifications to the variables l_{up}, l, p, i during the learning phase of the regular language represented by the minimal DFA from Figure 4.4 in the following example.

Example 4.3.4. In this example, an exemplifying run of the NL* algorithm will illustrate the evolution of the measure $\mathfrak{M}(\mathcal{T})$ that we associated with a table $\mathcal{T}$ in the proof of Theorem 4.3.3. Table 4.2 depicts the intermediate tables that we construct during the NL* run, as well as their associated measures in table $\mathfrak{M}$ (bottom of Table 4.2). The target automaton is taken from Figure 4.4.

Let us consider table $\mathcal{T}_0$ from Table 4.2. The first case that occurs while learning the regular language in mind is the detection of a counterexample for table $\mathcal{T}_0$. Adding a counterexample (here: *suff*(baa)) to the table results in the increase of variable l (by $k = 2$) as described in the proof above. After the table extension the number of different rows in the whole table raises from 1 to 3 as the new columns make all lines of the table distinct.

	$\mathcal{T}_0$	ε
$*$	ε	−
$*$	b	−
$*$	a	−

$\Rightarrow^{1.}_{ce}$

	$\mathcal{T}_1$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−

$\Rightarrow^{2.}_{ncl}$

	$\mathcal{T}_2$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
$*$	bb	−	−	−	−
$*$	ba	−	−	+	+

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
$*$	bb	−	−	−	−
$*$	ba	−	−	+	+
$*$	ab	−	−	−	−
$*$	aa	−	−	−	−

$\Rightarrow^{4.}_{ncl}$

	$\mathcal{T}_4$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
$*$	ba	−	−	+	+
$*$	bb	−	−	−	−
$*$	ab	−	−	−	−
$*$	aa	−	−	−	−
$*$	bab	−	−	−	−
$*$	baa	+	−	−	+

$\Rightarrow^{5.}_{ncl}$

	$\mathcal{T}_5$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
	ba	−	−	+	+
	baa	+	−	−	+
$*$	bb	−	−	−	−
$*$	ab	−	−	−	−
$*$	aa	−	−	−	−
$*$	bab	−	−	−	−
$*$	$baab$	−	−	−	+
$*$	$baaa$	+	−	−	−

$\Rightarrow^{6.}_{ncl}$

	$\mathcal{T}_6$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
	ba	−	−	+	+
	baa	+	−	−	+
$*$	$baab$	−	−	−	+
$*$	bb	−	−	−	−
$*$	ab	−	−	−	−
$*$	aa	−	−	−	−
$*$	bab	−	−	−	−
$*$	$baaa$	+	−	−	−
$*$	$baabb$	−	−	−	−
$*$	$baaba$	+	−	−	−

$\Rightarrow^{7.}_{ncl}$

	$\mathcal{T}_7$	ε	baa	aa	a
$*$	ε	−	+	−	−
$*$	b	−	−	+	−
$*$	a	−	−	−	−
	ba	−	−	+	+
	baa	+	−	−	+
$*$	$baab$	−	−	−	+
$*$	$baaa$	+	−	−	−
$*$	bb	−	−	−	−
$*$	ab	−	−	−	−
$*$	aa	−	−	−	−
$*$	bab	−	−	−	−
$*$	$baabb$	−	−	−	−
$*$	$baaba$	+	−	−	−
$*$	$baaab$	−	−	−	−
$*$	$baaaa$	−	−	−	−

$\Rightarrow^{8.}_{ncs}$

	$\mathcal{T}_8$	ε	baa	aa	a	aaa
$*$	ε	−	+	−	−	−
$*$	b	−	−	+	−	+
$*$	a	−	−	−	−	−
$*$	ba	−	−	+	+	−
	baa	+	−	−	+	−
$*$	$baab$	−	−	−	+	−
$*$	$baaa$	+	−	−	−	−
$*$	bb	−	−	−	−	−
$*$	ab	−	−	−	−	−
$*$	aa	−	−	−	−	−
$*$	bab	−	−	−	−	−
$*$	$baabb$	−	−	−	−	−
$*$	$baaba$	+	−	−	−	−
$*$	$baaab$	−	−	−	−	−
$*$	$baaaa$	−	−	−	−	−

$\Rightarrow^{9.}_{ce}$

	$\mathcal{T}_9$	ε	baa	aa	a	aaa	$baaba$	$aaba$	aba	ba
$*$	ε	−	+	−	−	−	+	−	−	−
$*$	b	−	−	+	−	+	−	+	−	−
$*$	a	−	−	−	−	−	−	−	−	−
$*$	ba	−	−	+	+	−	−	−	+	−
$*$	baa	+	−	−	+	−	−	−	−	+
$*$	$baab$	−	−	−	+	−	−	−	−	−
$*$	$baaa$	+	−	−	−	−	−	−	−	−
$*$	bb	−	−	−	−	−	−	−	−	−
$*$	ab	−	−	−	−	−	−	−	−	−
$*$	aa	−	−	−	−	−	−	−	−	−
$*$	bab	−	−	−	−	−	−	−	−	−
$*$	$baabb$	−	−	−	−	−	−	−	−	−
$*$	$baaba$	+	−	−	−	−	−	−	−	−
$*$	$baaab$	−	−	−	−	−	−	−	−	−
$*$	$baaaa$	−	−	−	−	−	−	−	−	−

	$\mathfrak{M}$	l_{up}	l	p	i	*Sum*
	$\mathcal{T}_0$	1	1	1	0	3
$\Rightarrow_{ce}$	$\mathcal{T}_1$	1	3	3	2	9
$\Rightarrow_{ncl}$	$\mathcal{T}_2$	2	4	4	4	14
$\Rightarrow_{ncl}$	$\mathcal{T}_3$	3	4	4	4	15
$\Rightarrow_{ncl}$	$\mathcal{T}_4$	4	5	5	5	19
$\Rightarrow_{ncl}$	$\mathcal{T}_5$	5	7	5	10	27
$\Rightarrow_{ncl}$	$\mathcal{T}_6$	6	7	5	10	28
$\Rightarrow_{ncl}$	$\mathcal{T}_7$	7	7	5	10	29
$\Rightarrow_{ncs}$	$\mathcal{T}_8$	7	7	6	9	29
$\Rightarrow_{ce}$	$\mathcal{T}_9$	7	7	7	9	30

Table 4.2: An NL* run and its corresponding measures

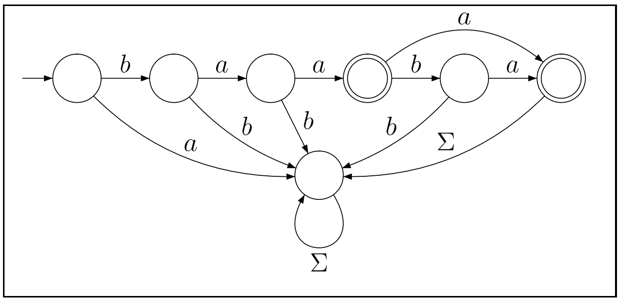


Figure 4.4: Minimal DFA for the (finite) regular language of Example 4.3.4

Simultaneously the number p of primes increases to 3 and the number of strict inclusion relations from 0 to 2, accordingly. Regarding table $\mathcal{T}_8$, we have $\mathfrak{M}(\mathcal{T}_8) = (7, 7, 6, 9)$ and $\mathfrak{M}(\mathcal{T}_9) = (7, 7, 7, 9)$. I.e., after adding the second counterexample, both variables, l and i, remain unchanged. However, p is indeed increased as described in the proof.

The case of a non-RFSA-closed table occurs, e.g., in table $\mathcal{T}_1$. As $row(b)$ has become a prime row but is not yet situated in the upper table, it has to be moved to U resulting in an increase of variable l_{up} and variable l by one, each. The newly added row $row(ba)$ constitutes a new prime for table $\mathcal{T}_2$ and hence, variable p is augmented by one. Due to this new row, we also get two more strict inclusions, yielding an increment of variable i by two.

The last possible case occurs in table $\mathcal{T}_7$. It is not RFSA-consistent because rows $row(b)$ and $row(ba)$ are in the covering relation ($row(b) \sqsubseteq row(ba)$) but their a-successors are incomparable. This conflict is resolved by adding the distinguishing string aaa to the set of columns V. The measure from table $\mathcal{T}_7$ to table $\mathcal{T}_8$ leaves variable l unchanged. As mentioned in the proof, in this case, variable i has to be decreased by at least one. A close look at the tables $\mathcal{T}_8$ convinces us: variable i is indeed decreased by one. At the same time, the number of primes p is increased. Row $row(ba)$, which was composed in table $\mathcal{T}_7$, now became prime.

The following schedule explicitly describes the whole run of NL* for the regular language represented by the DFA from Figure 4.4. During the run two intermediate models and the final canonical RFSA were calculated.

1) Found counterexample baa for current model $\mathcal{A}_{\mathcal{T}_0}$ (based on table $\mathcal{T}_0$).
2) RFSA-closedness violation: Trying to make $\mathcal{T}_1$ RFSA-closed with row: b
3) RFSA-closedness violation: Trying to make $\mathcal{T}_2$ RFSA-closed with row: a
4) RFSA-closedness violation: Trying to make $\mathcal{T}_3$ RFSA-closed with row: ba
5) RFSA-closedness violation: Trying to make $\mathcal{T}_4$ RFSA-closed with row: baa
6) RFSA-closedness violation: Trying to make $\mathcal{T}_5$ RFSA-closed with row: $baab$
7) RFSA-closedness violation: Trying to make $\mathcal{T}_6$ RFSA-closed with row: $baaa$
8) RFSA-consistency violation: Trying to obtain RFSA-consistency for table $\mathcal{T}_7$ by adding the distinguishing suffix aaa
9) Found counterexample $baaba$ for current model $\mathcal{A}_{\mathcal{T}_8}$ (based on table $\mathcal{T}_8$).
10) Table $\mathcal{T}_9$ is RFSA-closed and RFSA-consistent. Final model $\mathcal{H}$ can be calculated.

◇

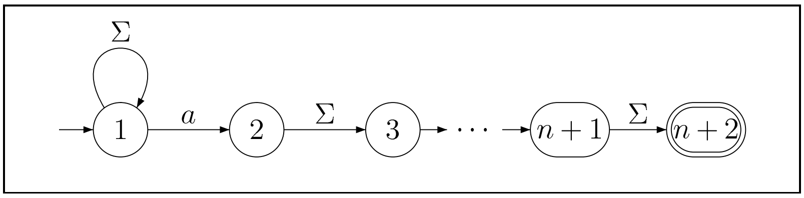


Figure 4.5: An NFA over $\Sigma = \{a, b\}$ accepting the regular language L_n with $n + 2$ states

$\mathcal{T}_0$	ε
ε	−
a	−
b	−

1) ⇒

$\mathcal{T}_1$	ε
ε	−
a	−
aa	−
aaa	+
b	−
ab	−
aab	+
$aaaa$	+
$aaab$	+

2) ⇒

$\mathcal{T}_2$	ε	a
ε	−	−
a	−	−
aa	−	+
aaa	+	+
b	−	−
ab	−	+
aab	+	+
$aaaa$	+	+
$aaab$	+	+

3) ⇒

$\mathcal{T}_3$	ε	a	ba
ε	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
b	−	−	−
ab	−	+	−
aab	+	+	−
$aaaa$	+	+	+
$aaab$	+	+	−

4) ⇒

$\mathcal{T}_4$	ε	a	ba
ε	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
ab	−	+	−
aab	+	+	−
b	−	−	−
$aaaa$	+	+	+
$aaab$	+	+	−
aba	+	−	+
abb	+	−	−
$aaba$	+	−	+
$aabb$	+	−	−

5) ⇒

$\mathcal{T}_5$	ε	a	ba
ε	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
ab	−	+	−
aab	+	+	−
$aaba$	+	−	+
$aabb$	+	−	−
b	−	−	−
$aaaa$	+	+	+
$aaab$	+	+	−
aba	+	−	+
abb	+	−	−
$aabaa$	−	+	+
$aabab$	−	+	−
$aabba$	−	−	+
$aabbb$	−	−	−

Table 4.3: Learning regular language L_2 employing L*

4.4 NL* vs. L*: an Example for an Exponential Gain

In this section we will present an extended example of the functioning of the NL* algorithm and show that there is an infinite family of regular languages for which NL* learns exponentially more succinct automata than its deterministic version L*.

Suppose we were given a two-letter alphabet $\Sigma = \{a, b\}$ and let $L_n \subseteq \Sigma^*$ be the language of words over Σ containing the letter a at the $(n+1)$-last position according to the regular expression $\Sigma^* \, a \, \Sigma^n$. Then, L_n is accepted by a minimal DFA $\mathcal{A}_n^*$ with 2^{n+1} states. Nevertheless, there are NFA (cf. Figure 4.5) with only $n + 2$ states accepting L_n. It is easy to see that there is even a canonical RFSA $\mathcal{R}_n$ of size $n + 2$ accepting L_n. In other words, $\mathcal{R}_n$ is exponentially more succinct than $\mathcal{A}_n^*$.

Now we exemplarily show how L_2, whose minimal DFA $\mathcal{A}_2^*$ is given in Figure 4.6, is learned by Angluin's L* algorithm and by our algorithm NL*. We start with a run of L*, which is illustrated in Table 4.3. Table $\mathcal{T}_0$ is closed and consistent but does not represent the intended automaton because, e.g., the word *aaa* is not accepted but contained in L_2. Hence, we add $\mathit{pref}(aaa)$ to U and $\mathit{pref}(aaa)\Sigma$ to $U\Sigma$. The result (after performing the necessary membership queries) is $\mathcal{T}_1$. This table is closed but not consistent ($\mathit{row}(a) = \mathit{row}(aa)$ but not $\mathit{row}(aa) = \mathit{row}(aaa)$). Thus, we add the column a and obtain $\mathcal{T}_2$, which is still not consistent leading to $\mathcal{T}_3$. After making the table closed, we obtain $\mathcal{T}_5$, which is consistent as well, and whose corresponding automaton (Figure 4.6) accepts L_2.

Now we present a run of NL* for inferring $L_2 = \Sigma^* a \Sigma^2$. It is depicted in Table 4.4. As mentioned earlier, rows with a preceding $*$-symbol are prime rows. The table $\mathcal{T}_0$ is RFSA-closed and RFSA-consistent but does not represent the intended automaton because the word *aaa* is not accepted by the intermediate hypothesis though contained in language

	$\mathcal{T}_0$	ε
$*$	ε	$-$
$*$	b	$-$
$*$	a	$-$

$\Rightarrow^{1.}_{ce}$

	$\mathcal{T}_1$	ε	aaa	aa	a
$*$	ε	$-$	$+$	$-$	$-$
$*$	b	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$

$\Rightarrow^{2.}_{ncl}$

	$\mathcal{T}_2$	ε	aaa	aa	a
$*$	ε	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	b	$-$	$+$	$-$	$-$
$*$	ab	$-$	$+$	$-$	$+$
	aa	$-$	$+$	$+$	$+$

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	aaa	aa	a
$*$	ε	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	ab	$-$	$+$	$-$	$+$
$*$	b	$-$	$+$	$-$	$-$
	aa	$-$	$+$	$+$	$+$
$*$	abb	$+$	$+$	$-$	$-$
	aba	$+$	$+$	$+$	$-$

$\Rightarrow^{4.}_{ncl}$

	$\mathcal{T}_4$	ε	aaa	aa	a
$*$	ε	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	ab	$-$	$+$	$-$	$+$
$*$	abb	$+$	$+$	$-$	$-$
$*$	b	$-$	$+$	$-$	$-$
	aa	$-$	$+$	$+$	$+$
	aba	$+$	$+$	$+$	$-$
$*$	$abbb$	$-$	$+$	$-$	$-$
$*$	$abba$	$-$	$+$	$+$	$-$

Table 4.4: Learning regular language L_2 employing NL*

L_2. We add aaa and all its suffixes to V, perform membership queries, and then obtain table $\mathcal{T}_1$, which is not RFSA-closed. We add a to U and continue. After solving two more RFSA-closedness violations, we finally obtain table $\mathcal{T}_4$ which is RFSA-closed and RFSA-consistent, and its corresponding automaton given in Figure 4.7 is the canonical RFSA for L_2. Notice that table $\mathcal{T}_4$ is not closed in the Angluin sense and hence, L* would continue adding strings to the upper part of the table.

Corollary 4.4.1 ([DLT04]). *There are infinite families of regular languages for which NL* infers exponentially more succinct finite-state automata than L*.*

Proof: For example, the language classes $\mathcal{L} = \{L_n \mid n \in \mathbb{N}\}$ and the set of complements $\overline{\mathcal{L}} = \{\overline{L_n} \mid n \in \mathbb{N}\}$, where $\overline{L_n} := \Sigma^* \setminus L_n$, are infinite families of regular languages for which the canonical RFSA are exponentially more succinct than the corresponding minimal DFA. □

Remark 4.4.2. One might be tempted to assume that there is a serious catch when choosing NFA as underlying computation model for inference algorithms. For DFA there exist efficient algorithms for minimization and verifying equivalence. An important statement has to be given concerning the equivalence tests for NFA, however. Even if the target regular language is given in terms of a deterministic finite-state automaton, the standard method of *subset construction*, transforming the intermediate NFA into (minimal) deterministic automata, just annihilates the advantage of having compact representations of regular languages. This method can however be circumvented by using the *antichain* approach by De Wulf et al. [WDHR06], which yields a practically usable method for checking the language-inclusion problem for NFA which, in most applications, turns out to perform better than the naïve approach by several orders of magnitude. ◇

The gain in terms of states when using NL* instead of L* seems to be highest when a regular language on basis of a regular expression containing many "*or*" connectives is learned. Intuitively this kind of connectives expresses nondeterministic behavior and can thus be exploited optimally when learning an NFA. The L* algorithm, however, has to resolve this nondeterminism which may result in an exponential state blowup.

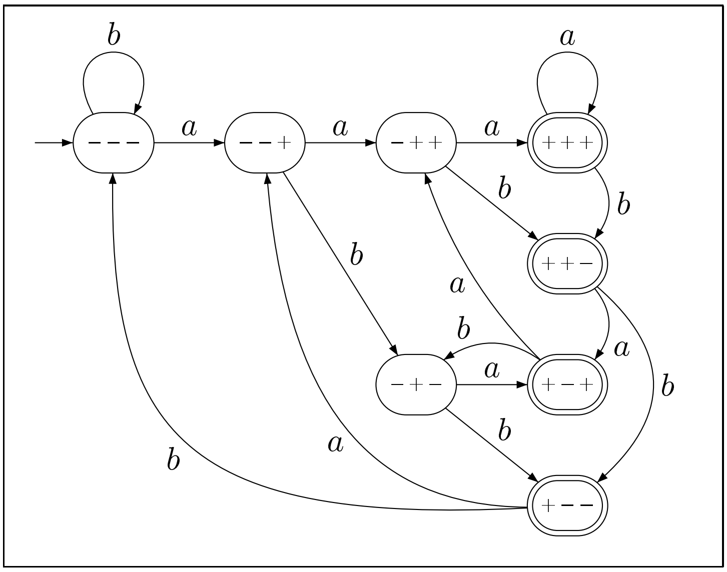


Figure 4.6: Minimal DFA $\mathcal{A}_2^*$ accepting language L_2 with 8 (= 2^{2+1}) states

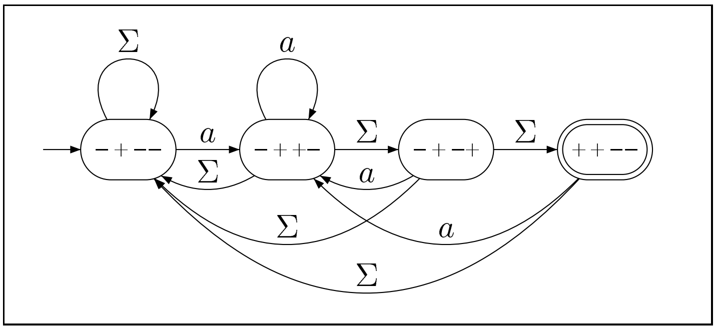


Figure 4.7: Canonical RFSA $\mathcal{R}_2$ accepting L_2 with 4 (= $2 + 2$) states

4.5 Other Approaches

We would like to mention two more approaches to define an active online learning algorithm for canonical RFSA, which appear to be reasonable at first sight, but, on closer inspection, turn out to be suboptimal in several respects.

4.5.1 A Straightforward but Inefficient Approach

Another possibility of on-the-fly exploiting inclusion relations between residual languages will be discussed in the following. We adopt the idea of checking for inclusion relations between rows in the table from the previous sections. To this end, we introduce a new notion of consistency, which we will call *strong RFSA-consistency.*

Definition 4.5.1 (Strong RFSA-Consistency). *A table $\mathcal{T} = (T, U, V)$ is called* strongly RFSA-consistent *(or strongly consistent, for short) if the following statements hold:*

(i) for all $u, u' \in U$ and $a \in \Sigma$, $row(u') \sqsubseteq row(u)$ implies $row(u'a) \sqsubseteq row(ua)$ and

(ii) for all $u \in U$ and $C \in 2^U \setminus \{\emptyset\}$: if $row(u) = \bigsqcup_{u' \in C} row(u')$ then

$$\textit{for all } a \in \Sigma : row(ua) = \bigsqcup_{u' \in C} row(u'a).$$

Intuitively, for a given table $\mathcal{T} = (T, U, V)$, strong RFSA-consistency checks *(i)* for all inclusion relations whether their successors are in this relation as well and, additionally, *(ii)* if for all possibilities C a row r could be composed of, the successor of r is composed of the successors of C. Replacing the original consistency check of the NL* algorithm with the new strong-consistency check yields an algorithm that will be usable to infer (canonical) RFSA. Nevertheless, this algorithm is, as the one presented in Section 4.1, highly inefficient. There may be exponentially many possibilities of sets of states composing a given state. Thus, the strong-consistency check has an exponential time complexity in the number of rows of U (i.e., in the number of states of the hypothesis) and can thus not be employed for deriving an efficient online learning algorithm tailored to RFSA.

Another idea is to proceed similar as in Angluin's algorithm and to add counterexamples and their prefixes to the set of rows U. But more importantly, we have to relax the notion of consistency such that, in contrast to our strong consistency defined above, the new version can be checked efficiently. Using such a notion, we might be able to obtain another efficient online algorithm for inferring canonical RFSA, which would be closely related to L* (and not L$^*_{\text{col}}$).

4.5.2 A More Elaborate but Non-terminating Approach

As the previous definition of strong-consistency turned out not to be checkable efficiently, in this subsection we define a new notion of consistency called *weak RFSA-consistency*. As opposed to the previous algorithm, the new notion of consistency will be efficiently testable, as we will not consider all possibilities of sets composing a state any more but only maximal ones.

We will now define this new notion of consistency called weak RFSA-consistency.

Definition 4.5.2 (Weak RFSA-Consistency). *A table $\mathcal{T} = (T, U, V)$ is called* weakly RFSA-consistent *(or weakly consistent, for short) if the following statements hold:*

(i) for all $u, u' \in U$ and $a \in \Sigma$, $row(u') \sqsubseteq row(u)$ implies $row(u'a) \sqsubseteq row(ua)$ and

(ii) for all $u \in U$ and $a \in \Sigma$: $row(ua) = \bigsqcup_{\{u' \mid row(u') \sqsubseteq row(u)\}} row(u'a)$.

An implementation of this approach very much resembles the algorithm from Table 4.1. Therefore, we refrain from listing the whole pseudocode as the only difference is the notion of consistency in lines 3 and 8 which have to be replaced by the concept of weak consistency and line 19, which has to be changed to $\mathcal{T} = (T', U \cup pref(w), V)$, as we now want to add counterexamples and their prefixes to rows.

On the first sight, an implementation of an algorithm based upon the definition of weak consistency appears to be reasonable. Unfortunately, we are in a situation where we can no longer prove termination of the underlying algorithm and, as it turns out when implementing this version, even if it constructs the correct canonical RFSA for most regular languages, there are indeed examples for which it does not terminate. Such an example—using the original definition of RFSA-consistency (cf. Definition 4.2.7 on page 49)—is given in Appendix B.4 on page 181.

The reason for non-termination is that certain states in the hypothesis automaton are not reachable by their access string. In the learning instance from Appendix B.4, for example, access string a represents state "$+-$", hence, word a should be accepted by the hypothesis. As shown in Figure B.6(b) on page 181, this is not the case. This entails that a might again be presented as counterexample to the learning algorithm. But as this row is already contained in the table, no helpful information is added to continue the learning procedure. The learning algorithm will again check the table for closedness and consistency properties and—because the old and the new table are identical—return the same hypothesis as before, yielding a non-terminating cycle in the algorithm.

4.6 UL*: Remodeling NL* for Learning Universal Automata

As adumbrated at the end of Section 4.4, NL* performs best if there are many "*or*" connectives in the underlying regular expression equivalent to the target language and, thus, much nondeterminism in the language to infer. Let us assume we were given such a regular language L for which NL* performs very well, i.e., with a high degree of nondeterminism. If we complement this language, we are confronted with a regular language $\overline{L}$ containing only a few "*or*" but many "*and*" connectives. Even if usually still better than L* concerning the number of states of the resulting automata, NL* will not operate well on such a language. A simple idea resolves this unpleasant observation. Instead of employing NL* for language $\overline{L}$, we complement the definition of the covering relation and change the automaton model from nondeterministic to universal automata, i.e., automata featuring "*and*"- instead of "*or*"-transitions, as defined in Definitions 2.3.19 and 2.3.21.

As before, we have to come up with a new notion of the former join operator (this will be the *intersection* operator) and of *prime* and *composed* rows but now in the setting of universal automata.

Definition 4.6.1 (Intersection Operator). *Let $\mathcal{T} = (T, U, V)$ be a table as usual. The* intersection $(r_1 \sqcap r_2) : V \to \{+, -\}$ *of two rows $r_1, r_2 \in Rows(\mathcal{T})$ is defined componentwise for each $v \in V$: $(r_1 \sqcap r_2)(v) := r_1(v) \sqcap r_2(v)$ where $- \sqcap - = + \sqcap - = - \sqcap + = -$ and $+ \sqcap + = +$.*

As for the case of RFSA, note that the intersection operator is associative, commutative, and idempotent, yet that the intersection of two rows is *not* necessarily a row of table $\mathcal{T}$ again.

Definition 4.6.2 ($\sqcap$-Composed and $\sqcap$-Prime Rows). *Let $\mathcal{T} = (T, U, V)$ be a table. A row $r \in Rows(\mathcal{T})$ is called* $\sqcap$-composed *if there are rows $r_1, \ldots, r_n \in Rows(\mathcal{T}) \setminus \{r\}$ such that $r = r_1 \sqcap \ldots \sqcap r_n$. Otherwise, r is called* $\sqcap$-prime. *The set of $\sqcap$-prime rows in $\mathcal{T}$ is denoted by $Primes^{\sqcap}(\mathcal{T})$. Moreover, we let $Primes^{\sqcap}_{\mathrm{upp}}(\mathcal{T}) = Primes^{\sqcap}(\mathcal{T}) \cap Rows_{\mathrm{upp}}(\mathcal{T})$.*

As before, we call a row $r \in Rows(\mathcal{T})$ of a table $\mathcal{T}$ $\sqcap$-*covered* by row $r' \in Rows(\mathcal{T})$, denoted by $r \sqsubseteq r'$, if for all $v \in V$, $r(v) = -$ implies $r'(v) = -$. If moreover $r' \neq r$, then r is *strictly covered by* r', denoted by $r \sqsubset r'$.

Note that r may be $\sqcap$-covered by r' and, both, r and r' are $\sqcap$-prime. A $\sqcap$-composed row covers all the $\sqcap$-primes it is $\sqcap$-composed of.

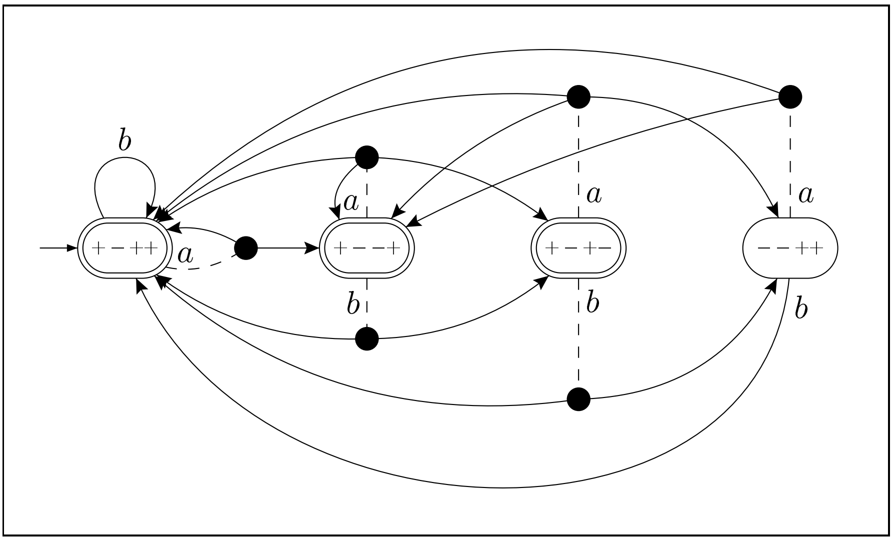
The following definitions of *RUFA-closedness* and *RUFA-consistency* are similar to their RFSA equivalent.

Definition 4.6.3 (RUFA-Closedness). *A table $\mathcal{T} = (T, U, V)$ is called* RUFA-closed *if, for each $r \in Rows_{\mathrm{low}}(\mathcal{T})$, $r = \bigsqcap\{r' \in Primes^{\sqcap}_{\mathrm{upp}}(\mathcal{T}) \mid r \sqsubseteq r'\}$.*

Definition 4.6.4 (RUFA-Consistency). *A table $\mathcal{T} = (T, U, V)$ is called* RUFA-consistent *if, for all $u, u' \in U$ and $a \in \Sigma$, $row(u') \sqsubseteq row(u)$ implies $row(u'a) \sqsubseteq row(ua)$.*

	$\mathcal{T}_4$	ε	aaa	aa	a
$*$	ε	$+$	$-$	$+$	$+$
$*$	a	$+$	$-$	$-$	$+$
$*$	ab	$+$	$-$	$+$	$-$
$*$	abb	$-$	$-$	$+$	$+$
$*$	b	$+$	$-$	$+$	$+$
	aa	$+$	$-$	$-$	$-$
	aba	$-$	$-$	$-$	$+$
$*$	$abbb$	$+$	$-$	$+$	$+$
$*$	$abba$	$+$	$-$	$-$	$+$

(a)



(b)

Figure 4.8: Final table $\mathcal{T}_4$ and corresponding canonical RUFA for language $\overline{L_2}$

We are now prepared to define the universal automaton of a RUFA-closed and RUFA-consistent table $\mathcal{T}$.

Definition 4.6.5 (UFSA of a Table)**.** *For a table $\mathcal{T} = (T, U, V)$ that is RUFA-closed and RUFA-consistent, we define an UFSA $\mathcal{U}_\mathcal{T} = (Q, Q_0, \delta, F)$ over alphabet Σ by:*

- $Q = \mathit{Primes}^{\sqcap}_{\mathrm{upp}}(\mathcal{T})$,
- $Q_0 = \{r \in Q \mid \mathit{row}(\varepsilon) \sqsubseteq r\}$,
- $\delta(\mathit{row}(u), a) = \{r \in Q \mid \mathit{row}(ua) \sqsubseteq r\}$ *for* $u \in U$ *with* $\mathit{row}(u) \in Q$ *and* $a \in \Sigma$, *and*
- $F = \{r \in Q \mid r(\varepsilon) = +\}$.

All theoretical results from Sections 4.2, 4.3 and 4.4 also hold in the setting of RUFA and their proofs are analog. Thus, we only mention the most important results being transferred from the NL* learning sections.

Corollary 4.6.6. *Let $\mathcal{T}$ be a table that is RUFA-closed and RUFA-consistent and let $\mathcal{U}_\mathcal{T}$ be consistent with $\mathcal{T}$ (as defined in Definition 4.2.15). Then, $\mathcal{U}_\mathcal{T}$ is a canonical RUFA.*

Corollary 4.6.7. *Let n be the number of states of the minimal DFA $\mathcal{A}^*$ for a given regular language $L \subseteq \Sigma^*$. Let m be the length of the longest counterexample returned by the equivalence test (or 1 if the equivalence test always succeeds). Then,* UL* *returns after at most $\mathcal{O}(n^2)$ equivalence queries and $\mathcal{O}(m|\Sigma|n^3)$ membership queries the canonical RUFA $\mathcal{U}(L)$.*

Finally, we state that learning RUFA using the new UL* algorithm is sensible.

Corollary 4.6.8. *There are infinite families of regular languages for which* UL* *infers an exponentially more succinct automaton than L*.*

Proof: The language class $\mathcal{L} = \{L_n \mid n \in \mathbb{N}\}$ from Section 4.4 (and the set of complements of $\mathcal{L}$, $\overline{\mathcal{L}} = \{\overline{L_n} \mid n \in \mathbb{N}\}$) are infinite families of regular languages for which the canonical RUFA (and RFSA) are exponentially more succinct than the corresponding minimal DFA. □

Example 4.6.9. In Figure 4.8 an example universal automaton for the regular language $\overline{L_2}$ is presented. On the left-hand side, the final table $\mathcal{T}_4$ inferred by UL* is depicted, whereas the right-hand side shows the RUFA derived from table $\mathcal{T}_4$. In fact, it is even the canonical RUFA for $\overline{L_2}$. Like the canonical RFSA for language L_2 (cf. Figure 4.7) this canonical RUFA is also exponentially more succinct than its equivalent minimal DFA (cf. minimal DFA from Figure 4.6 with inverted final states). ◇

In general, if we were interested in the smallest possible model (i.e., canonical RFSA or canonical RUFA) it would be reasonable to simultaneously run the two variants NL* and UL* in order to obtain the smaller representation of the target regular language. Of course, in the worst case all three representations of the target, namely minimal DFA, canonical RFSA, and canonical RUFA, are of equal size, but in the best case (one of) the latter two might be exponentially more succinct than their deterministic counterpart.

4.7 Experiments

To evaluate the practical performance of NL* (and UL*), we compare our learning algorithms with Angluin's L* algorithm, and its modification, called L$^*_{\text{col}}$, from Subsection 3.3.1. As NL* and UL* are arguably similar in spirit to L$^*_{\text{col}}$, a comparison with this algorithm seems more fair.

To obtain maximally significant statistical data, we implemented all four learning algorithms and let them execute the same set of samples. Following [DLT04], we randomly generated large sets of regular expressions over different sizes of alphabets and let the algorithms infer the languages induced by the input regular languages. The derivation of the sample sets used in our experiments is described in the following paragraph.

4.7.1 Derivation of Sample Sets

As proposed in [DLT04], we randomly generated large sets of regular expressions over different sizes of alphabets Σ. By means of a context-free grammar for inducing regular expressions, we iteratively constructed derivations of randomly drawn length l (i.e., number of production rule applications). To this end, we randomly drew a number representing the number of rule applications for the regular grammar. Then l productions of this grammar were randomly chosen according to fixed probabilities for the three operators *concatenation* $(\cdot, 0.556)$, *choice* $(+, 0.278)$ and *Kleene star* $(^*, 0.166)$. As result, we got an object consisting only of non-terminals and linked by l regular expression operators. To obtain the final regular expression, all non-terminal symbols of the derivation were replaced by a letter from the corresponding alphabet according to a uniform distribution over Σ.

For the statistics enclosed in this thesis, we generated several sets of regular expressions for different alphabet sizes ($|\Sigma| \in \{2, 3\}$). To assure that the minimal DFA equivalent to the samples generated were all within a certain range $\mathfrak{R}$ of state numbers, we created regular expressions, transformed them to their minimal DFA equivalent, and tested the number of states. We only added them to the final test set if their state number was within range $\mathfrak{R}$, and the sample size was not already present in the test set. This procedure was iterated four times and terminated after we had four representatives of each state size. For the statistics of this section, we only compiled regular expressions which had equivalent minimal DFA between 1 and 200 states. These sets were fed to the three learning algorithms L^*, $\mathrm{L}^*_{\mathrm{col}}$ and NL^* and the results stored. To evaluate the algorithms' performances, we recorded the sizes of the hypotheses and the numbers of membership and equivalence queries, respectively.

In the next paragraph, we summarize the results obtained from our experiments on 2- and 3-letter alphabets. The results are based on approximately 3200 sample regular expressions, each.

4.7.2 Statistical Results

We now present the statistical results extracted from the test set mentioned previously. In the first subsection, we compare L^*, $\mathrm{L}^*_{\mathrm{col}}$, and NL^* to get an impression on the feasibility of our new approach. A similar comparison is accomplished in the second paragraph where all languages from the test set and, additionally, their complements are inferred using $\mathrm{L}^*_{\mathrm{col}}$, NL^*, and UL^*.

The results obtained in this section are in line with the results of [DLT04], though our results might be better, as NL^* is able to infer canonical RFSA whereas DeLeTe2 infers automata of size between canonical RFSA and equivalent minimal DFA. Furthermore, Denis et al. performed their analyses on basis of two additional test sets containing randomly drawn DFA and randomly drawn NFA, respectively. As randomly generated DFA do not seem to contain many inclusion relations, the canonical RFSA of these languages were not much smaller than—or even of equal size as—the equivalent minimal DFA. In case of the NFA test set, however, results turned out to be similarly positive as in the case of randomly drawn regular expressions.

Statistics: L^*, $\mathrm{L}^*_{\mathrm{col}}$, NL^*

For the diagrams in this section, we generated a set of 3180 regular expressions, resulting in minimal DFA with sizes ranging between 1 and 200 states. These DFA were given to the learning algorithms such that membership and equivalence queries could be answered according to these automata. To evaluate the algorithms' performances, we measured, for each algorithm and input regular expression, the *number of states of the final automaton* (canonical RFSA or minimal DFA) and the *number of membership (resp. equivalence) queries* to infer it.

Figure 4.9 compares the number of states of the automata learned by L^* (or equivalently, $\mathrm{L}^*_{\mathrm{col}}$) (i.e., minimal DFA) and NL^* (i.e., canonical RFSA) for 2- and 3-letter alphabets. The number of states of the automata learned by NL^* is considerably smaller than that of L^* and $\mathrm{L}^*_{\mathrm{col}}$, confirming the results of [DLT04]. More importantly, in practice, the actual sizes of canonical RFSA compared to equivalent minimal DFA seem to follow an exponential gap.

Figure 4.10 juxtaposes the number of membership queries of the three algorithms. Note that for automata of size larger than 40 states, NL^* seems to need increasingly less membership queries than the other two algorithms for inferring deterministic automata. Moreover, we see that in the case of membership queries the basic version of Angluin performs much better than its extended version $\mathrm{L}^*_{\mathrm{col}}$. This, however, is not the case if one considers the number of equivalence queries necessary to infer the automata (cf.

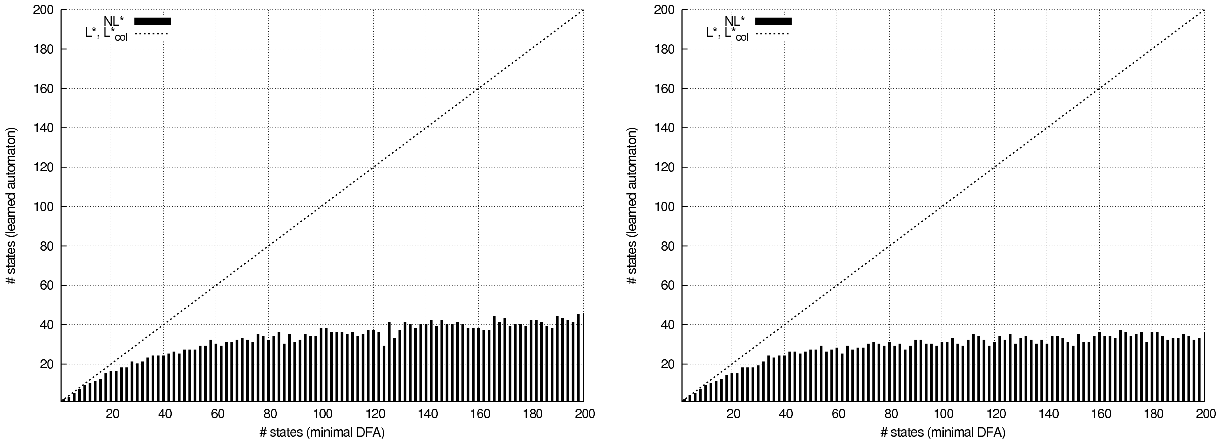


Figure 4.9: Number of states of minimal DFA and canonical RFSA ($|\Sigma| \in \{2, 3\}$)

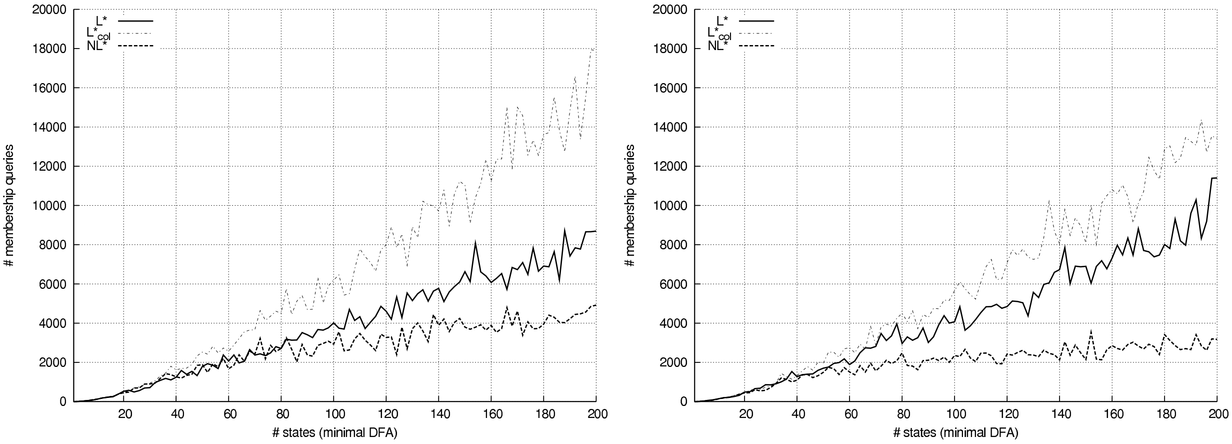


Figure 4.10: Number of membership queries ($|\Sigma| \in \{2, 3\}$)

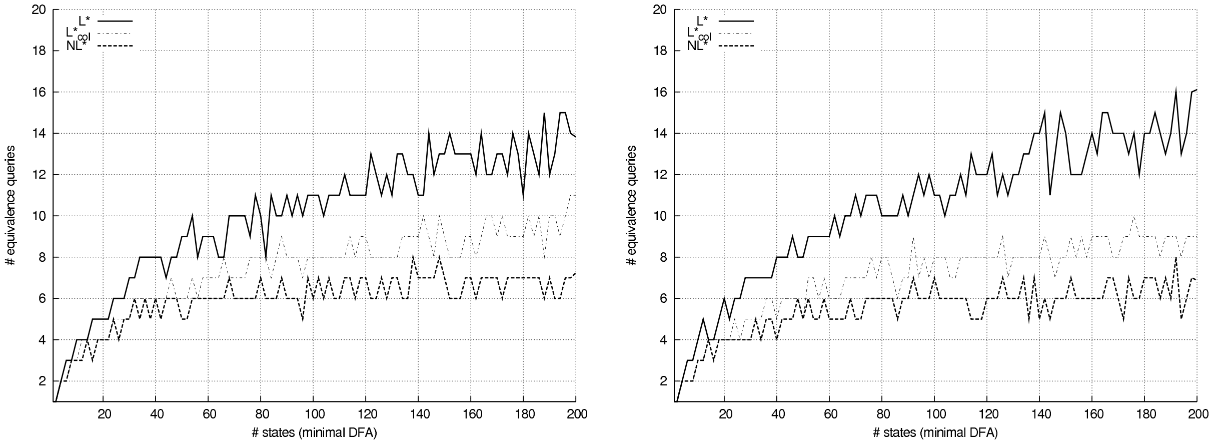


Figure 4.11: Number of equivalence queries ($|\Sigma| \in \{2, 3\}$)

NL^* and L^*	Won	Lost	Tie
States	95.78%	0.0%	4.22%
Membership queries	77.04%	22.01%	0.95%
Equivalence queries	89.64%	2.24%	8.12%

NL^* and $\mathrm{L}^*_{\mathrm{col}}$	Won	Lost	Tie
States	95.78%	0.0%	4.22%
Membership queries	88.85%	7.87%	3.28%
Equivalence queries	65.10%	13.32%	21.58%

Table 4.5: Comparing NL^* to L^* and $\mathrm{L}^*_{\mathrm{col}}$ (2-letter alphabet)

Figure 4.11). Though NL^* still performs far better than both, L^* and $\mathrm{L}^*_{\mathrm{col}}$, the extended version of Angluin now behaves nicer. In almost all cases it needs less equivalence queries than L^*. In many application areas, equivalence queries are extremely costly. Hence, for inferring deterministic automata, $\mathrm{L}^*_{\mathrm{col}}$ will in many cases be of high interest and preferable to the basic version L^*.

The above results are in contrast with the theoretical result we obtained in Theorem 4.3.3 on page 55. This suggests that the upper bound derived in Section 4.3 is not tight. The experiments we performed point out the clear predominance of NL^* over L^* and $\mathrm{L}^*_{\mathrm{col}}$ as long as the user is not dependent on a deterministic target model. Hence, NL^* might be of great interest, e.g., in the field of formal verification.

To get a numerical impression of "which algorithm is superior to which", consider the Tables 4.5 and 4.6. They emphasize the results mentioned above but also show that, in all cases, there is a significant number of wins of NL^* over the other two algorithms.

NL^* and L^*	Won	Lost	Tie
States	95.91%	0.0%	4.09%
Membership queries	81.92%	16.95%	1.13%
Equivalence queries	90.16%	2.20%	7.64%

NL^* and $\mathrm{L}^*_{\mathrm{col}}$	Won	Lost	Tie
States	95.91%	0.0%	4.09%
Membership queries	89.71%	7.08%	3.21%
Equivalence queries	64.34%	14.06%	21.60%

Table 4.6: Comparing NL^* to L^* and $\mathrm{L}^*_{\mathrm{col}}$ (3-letter alphabet)

The tables' entries have to be understood as follows: the column "Won" describes the number of times in our experiments where NL^* was superior to L^* or $\mathrm{L}^*_{\mathrm{col}}$, respectively, i.e., whenever the difference between number of states (or membership- or equivalence queries) of the automaton derived by $\mathrm{L}^*/\mathrm{L}^*_{\mathrm{col}}$ and by NL^* was positive. Similarly, column "Lost" describes when this number is negative. In case the difference is equal to 0 we have a "Tie". The same numbers were calculated for membership- and equivalence queries and are depicted in lines 2 and 3 of the corresponding tables.

Note that we also performed experiments with larger size alphabets ($|\Sigma| \in \{10, 20\}$). While currently we cannot make any general statement regarding membership and equivalence queries, concerning the number of states, these experiments show that even in the case of larger alphabets the exponential gap seems to hold though the curves get closer.

Learning Languages and Their Complements: $\mathrm{L}^*_{\mathrm{col}}$, NL^*, UL^*

In this section we want to demonstrate the problem that came up in Section 4.4 and was the reason for considering RUFA. If given a regular language L for which NL^* behaves very nicely, i.e., the result is substantially smaller than the minimal DFA for L, it might be the case that for the complement $\overline{L}$ of this language NL^* behaves much worse. In contrast, UL^* might work substantially better. Thus, in this paragraph we oppose the three algorithms ($\mathrm{L}^*_{\mathrm{col}}$, NL^* and UL^*) in order to learn the languages from another test set generated along the lines of the previous subsection as well as their complements. As it turns out,

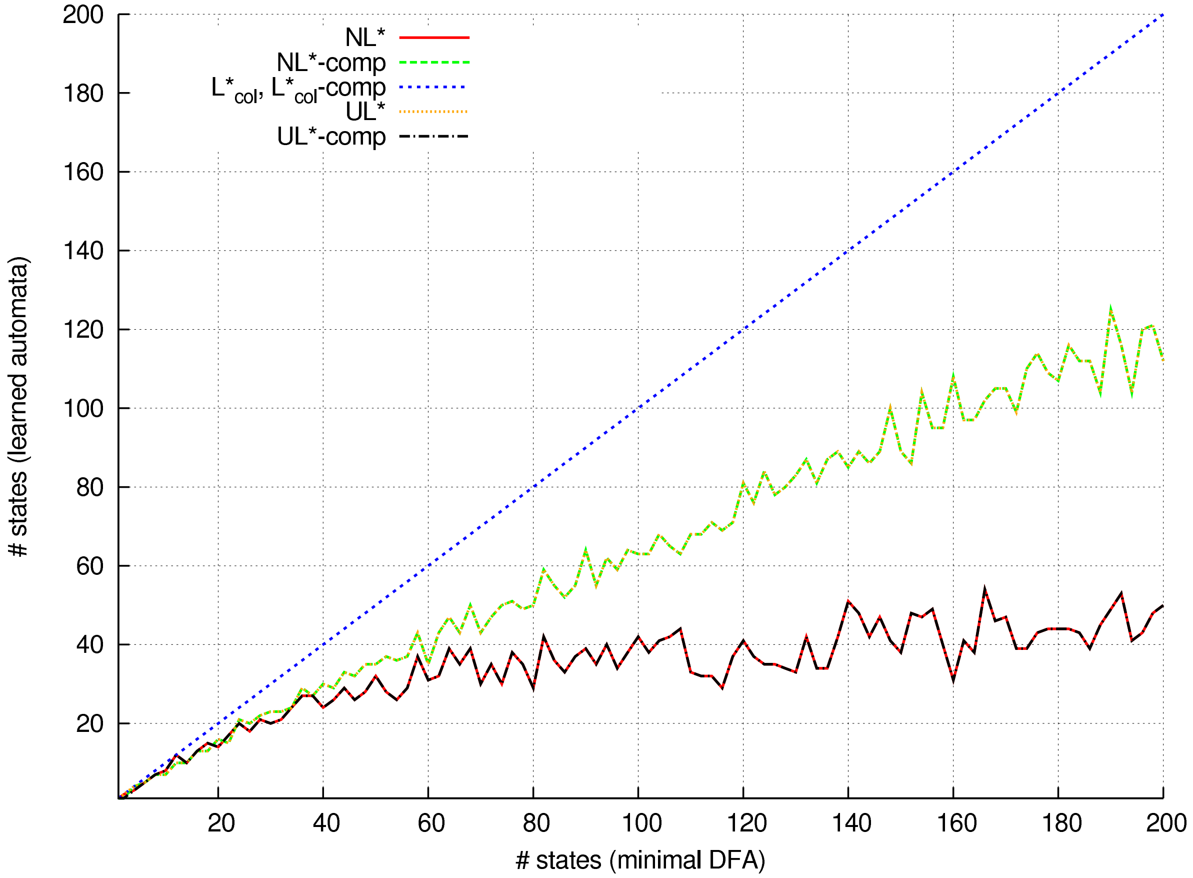


Figure 4.12: Number of states ($|\Sigma| = 2$)

either of the algorithms NL* and UL* performs very well in one of the two learning tasks. This observation is true for all three measures, i.e., number of states of the hypothesis, number of membership queries, and equivalence queries to obtain the hypothesis. But note moreover, that even in the case one of the two algorithms performs worse than the other, the number of states is still considerably smaller than that of the equivalent minimal DFA. In Figures 4.12, 4.13 and 4.14 the number of states, the number of membership queries and the number of equivalence queries for the three learning algorithms are opposed to one another. It can obviously be seen that for one of the inputs L and $\overline{L}$ always either NL* or UL* wins. Therefore—assuming that the hypothesis model (i.e., DFA, NFA, UFSA) is of no concern—we proposed, if given a regular language L', to let NL* and UL* run in parallel and take the smallest hypothesis as result. This combined learning algorithm still is a polynomial time learning algorithm, yielding the smallest of the three models.

4.8 Lessons Learned

Subsequently, we want to summarize the main properties characterizing our new active online learning algorithm NL*. We now give a brief discussion of its advantages and disadvantages with respect to other learning algorithms. Note that most considerations made in this section also apply to the UL* algorithm introduced in Section 4.6.

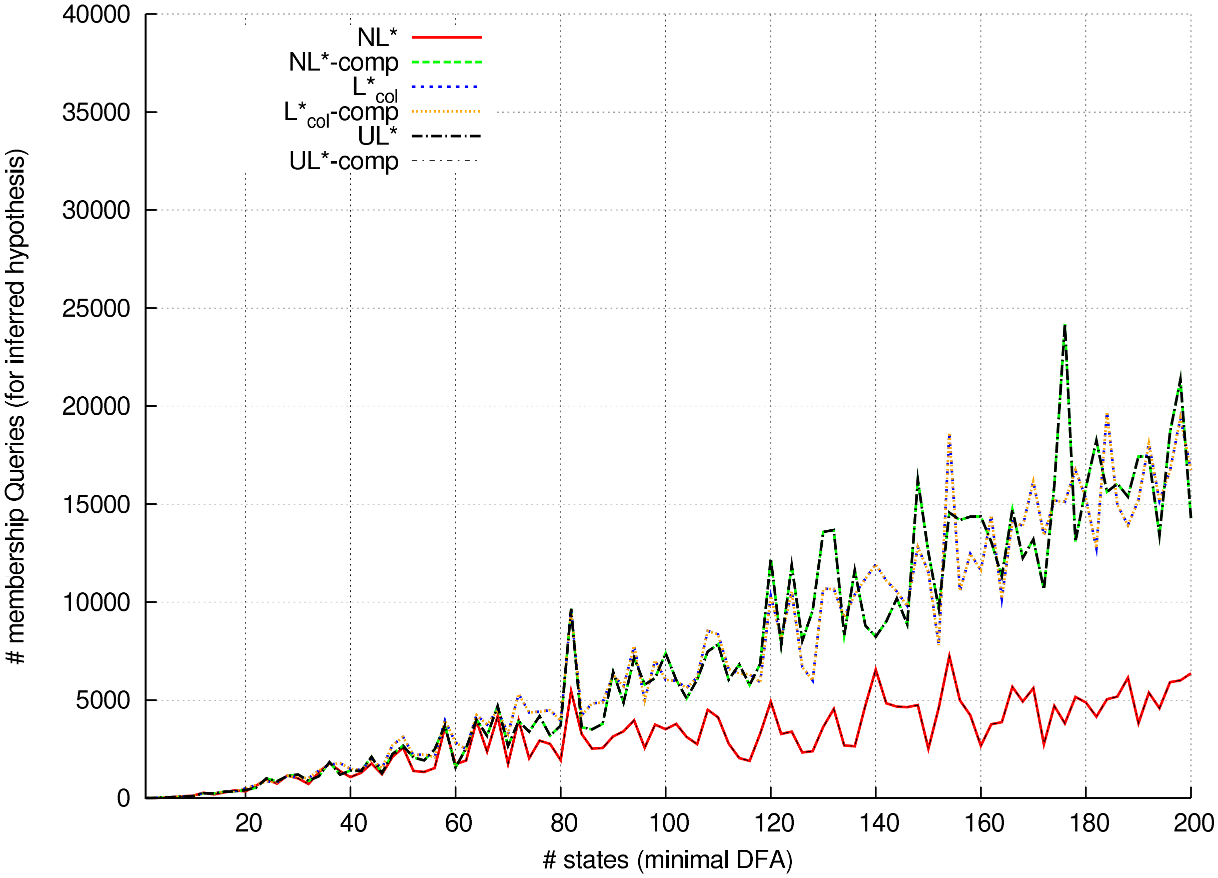


Figure 4.13: Number of membership queries ($|\Sigma| = 2$)

4.8.1 Why Choosing NL*?

Fist of all, we summarize the main characteristics of NL*. NL* is an active online learning algorithm for deriving a subclass of NFA, namely canonical RFSA, in the style of Angluin's famous L* algorithm (or, more precisely, its variant $\mathrm{L}^*_{\mathrm{col}}$). We showed that it is efficient regarding its theoretical complexity and, more importantly, performs well in practice. Moreover, the notorious problem of checking the equivalence between two NFA diminishes, as equivalence queries can be checked (more) efficiently using the antichain approach for checking language inclusion for NFA introduced in [WDHR06]. As we do not always get additional states after answering an equivalence query and submitting a new counterexample, the proof for termination of NL* turned out to be much more involved than the proof for the corresponding result for L* where it is known that each counterexample implies a growth of the state set by at least one.

4.8.2 Disadvantages

In this paragraph we like to mention several drawbacks a user might perceive when using NL*. Firstly, as we are learning canonical RFSA, which are a proper subset of NFA, this model might, due to the nondeterminism contained, not always be the desired synthesis model. Sometimes NL* needs more membership queries or equivalence queries than L* and $\mathrm{L}^*_{\mathrm{col}}$. This peculiarity goes hand in hand with the theoretical results we obtained for the time complexity for NL*, which is marginally higher than for L*. As shown

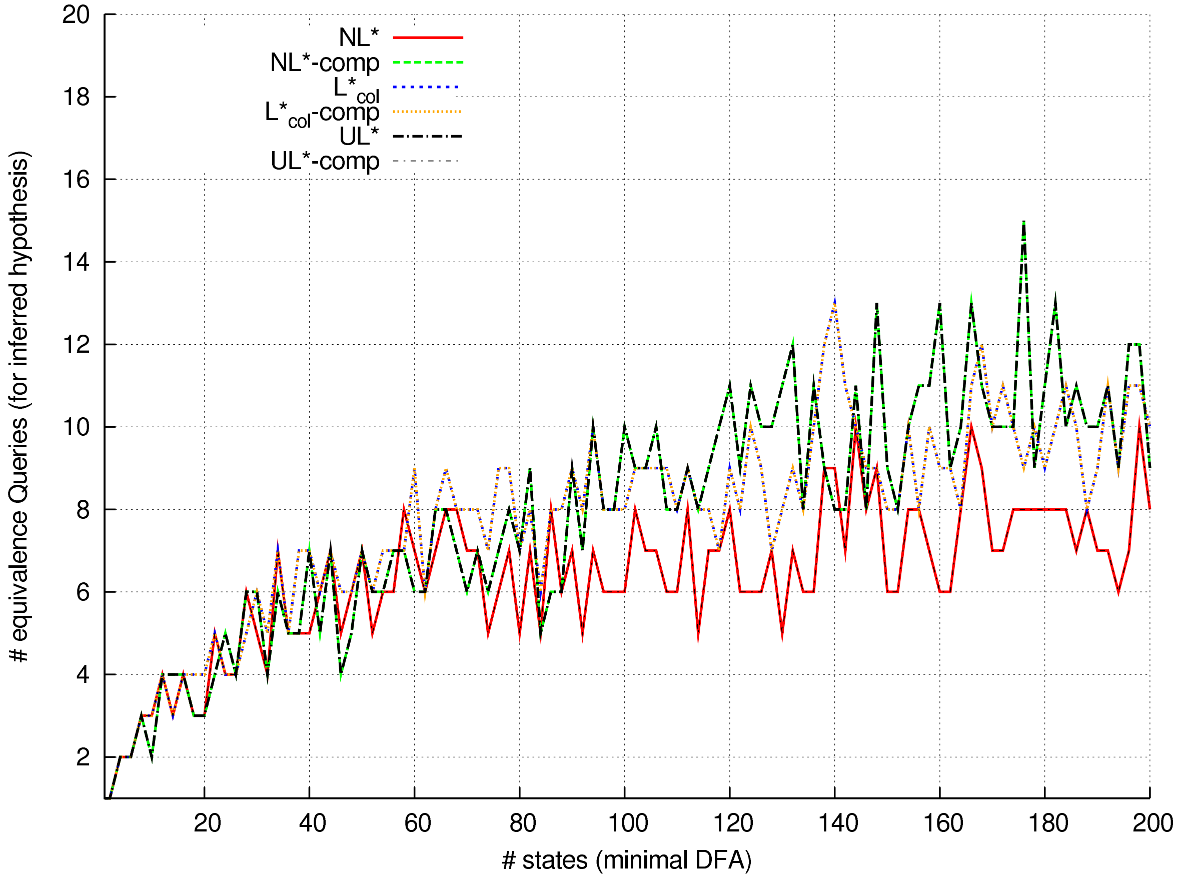


Figure 4.14: Number of equivalence queries ($|\Sigma| = 2$)

in the previous section, however, in the vast majority of cases NL* outperformed its deterministic versions L* and L^*_{col} by far if randomly drawn NFA or regular expressions are used as representation for the target regular languages. As shown in [DLT04] the gain drastically diminishes if minimal DFA are used as representations. Note that even if the alphabet size increases the exponential gap between the number of states of the minimal DFA and canonical RFSA seems to hold (if NFA or regular expressions are chosen as representation). Nevertheless, results turn out to be best when using small alphabets and if NFA are employed as representation.

Another property that is characteristic for NL* is that intermediate hypotheses need neither be canonical RFSA nor RFSA at all. As, however, the final model is always a canonical RFSA this fact might be of minor relevance to the user.

4.8.3 Advantages

After discussing some aspects of NL* which might be regarded problematic in some application areas, we now turn our interest to the advantages which accompany NL*. Assuming that the user is satisfied with a nondeterministic target model, NL* features several benefits. As we showed in Section 4.4, there are infinite families of regular languages which are recognized by canonical RFSA, which can be inferred using NL*, and these canonical RFSA are exponentially more succinct than their corresponding minimal DFA. For many applications areas, such as formal verification, small automata are desirable and, hence, NL* becomes the algorithm of choice. Our observations and experiences so far tell us that the unpleasant characteristics mentioned above do not occur in most cases and that, usually, even the inverse is true in the majority of cases. A further advantage, which distinguishes NL* from other learning algorithms for inferring nondeterministic automata is, the following: in NL* the final hypothesis for a regular language will always be a canonical RFSA. This is neither the case for any existing (passive) offline (i.e., the DeLeTe2 algorithm, cf. Subsection 3.2.2) nor active online algorithm (i.e., the LA algorithm, cf. Subsection 3.3.2) for inferring NFA. All in all, the advantages of NL* are manifold, and we are hopeful that it will be usable in many research areas like verification, e.g., in the setting of regular model checking [HV05], pattern matching [Yok95], or even in biological settings and bioinformatics, e.g., because according to [CF03] NFA are better suited to represent structure like "gaps" in genomic, or for characterizing protein families [CK06] where the classical learning algorithms like RPNI perform very badly.

4.9 Beyond Nondeterministic- and Universal Automata

To conclude this chapter, we take a glimpse beyond the learning of RFSA and RUFA. A model unifying the advantages of both worlds, i.e., having nondeterministic and universal transitions at the same time, could be interesting to regard for learning purposes. Alternating automata feature this property and are an alternative representation of regular languages. It is known that for a given regular language a corresponding minimal alternating automaton can be exponentially more succinct than an equivalent minimal NFA or UFSA and even doubly exponentially more succinct than the corresponding minimal DFA. Hence, it would be interesting to define the notion of *canonical residual alternating finite-state automata* (canonical RAFA, for short) as follows: a canonical RAFA for a regular language L is an alternating automaton such that every composed state of the equivalent minimal DFA can be represented as unions and intersections of prime residual languages of L. With this notion of canonical RAFA, we could try to learn the class by means of an Angluin-style learning algorithm. So far, we were only able to define an exponential time algorithm which, given a table $\mathcal{T}$, tries to find the minimal solution (i.e., a solution with a minimal number of prime rows) by testing all possibilities of composing a row out of other rows in order to find a minimal set of prime states from which residual languages are "constructible" (using union and intersection of prime residuals). This algorithm is obviously not efficient.

It is left open, whether the class of RAFA is learnable in polynomial time from an online learning algorithm in the spirit of L* or NL*.

5 Learning Congruence-Closed Languages

We now derive a schema improving the memory consumption of table-based learning algorithms like L^*, NL^* and UL^*. It groups words into equivalence classes so that they can be stored efficiently without the need of memorizing all words from Angluin's table, explicitly. The intention is to reduce the amount of memory necessary for storing Angluin's table. Instead of storing all elements of a class of congruent words, only one representative of each class, a *normal form*, will be recorded in the table. This approach amounts to merging rows and columns for congruent prefixes and suffixes, respectively, and leads to a substantial reduction of the table size as will be shown experimentally in Sections 6.3 and 9.2, where we apply this approach in the setting of message sequence charts.

Let Σ be an alphabet, $\mathcal{D} \subseteq \Sigma^*$ be a *domain*, and $\approx_{\mathcal{D}}$ be a congruence relation over $\mathcal{D}$. In order to represent congruence classes, we introduce a normal form for both prefixes and suffixes of words over $\mathcal{D}$. Consider a lexicographic (i.e., strict total) ordering on Σ, which is extended to words over Σ. For the rest of this chapter we will employ the length-lexicographical order $<_{\mathrm{lex}}$ from Definition 2.2.2. Let $\mathit{pnf}, \mathit{snf} : \Sigma^* \to \Sigma^* \uplus \{\bot\}$. The function pnf assigns to a word $w \in \mathit{pref}(\mathcal{D})$ the minimal word wrt. $<_{\mathrm{lex}}$ that is equivalent to w (to be made precise below). To words that are not in $\mathit{pref}(\mathcal{D})$, pnf assigns an element that is not from $\mathit{pref}(\mathcal{D})$ (the existence of such an element is guaranteed by the auxiliary symbol $\bot$). Analogously, the mapping snf assigns to a word $w \in \mathit{suff}(\mathcal{D})$ its normal form, i.e., the minimum (wrt. $<_{\mathrm{lex}}$) among all equivalent words, and it associates with every other word an element that is not from $\mathit{suff}(\mathcal{D})$ (its existence is again ensured by the character $\bot$). We define the concatenation with $\bot$ to be *annihilating*, i.e., for all $w \in \Sigma^* \uplus \{\bot\}$: $\bot \cdot w = w \cdot \bot = \bot$. The precise definition of the normal forms now goes as follows. Let $w \in \Sigma^*$:

- If $w \in \mathit{pref}(\mathcal{D})$, then we set $\mathit{pnf}(w) := \min_{<_{\mathrm{lex}}}\{w' \in \mathit{pref}(\mathcal{D}) \mid \exists v \in \Sigma^*\colon wv \approx_{\mathcal{D}} w'v\}$ where $\min_{<_{\mathrm{lex}}}$ returns the minimum of a given set wrt. $<_{\mathrm{lex}}$. Otherwise, let $\mathit{pnf}(w)$ be such that $\mathit{pnf}(w) \notin \mathit{pref}(\mathcal{D})$. In this case, we can also always choose the $\bot$ symbol.

- If $w \in \mathit{suff}(\mathcal{D})$, then we set $\mathit{snf}(w) := \min_{<_{\mathrm{lex}}}\{w' \in \mathit{suff}(\mathcal{D}) \mid \exists u \in \Sigma^*\colon uw \approx_{\mathcal{D}} uw'\}$. Otherwise, $\mathit{snf}(w) \notin \mathit{suff}(\mathcal{D})$. In this case, we can also always choose the $\bot$ symbol.

Note that $\mathit{pnf}(\varepsilon) = \mathit{snf}(\varepsilon) = \varepsilon$. The mappings pnf and snf are canonically extended to sets $L \subseteq \Sigma^*$, i.e., $\mathit{pnf}(L) = \bigcup_{w \in L} \mathit{pnf}(w)$ and $\mathit{snf}(L) = \bigcup_{w \in L} \mathit{snf}(w)$. We assume in the following that both pnf and snf are computable.

It is crucial for the application of normal forms that a given domain $\mathcal{D}$ satisfies, for all $u, v, u', v' \in \Sigma^*$, the following properties:

$$\text{If } uv \notin \mathcal{D}, \text{ then } u \notin \mathit{pref}(\mathcal{D}) \text{ or } v \notin \mathit{suff}(\mathcal{D}). \tag{*}$$

If $uv \in \mathcal{D}$ and $u'v' \approx_{\mathcal{D}} uv'$, then $u'v \in \mathcal{D}$. (**)

If $uv \in \mathcal{D}$ and $u'v' \approx_{\mathcal{D}} u'v$, then $uv' \in \mathcal{D}$. (***)

Assuming these properties as valid for a concrete domain $\mathcal{D}$, it will suffice to use normal forms when building a table in the extension of Angluin's algorithm, which may result in significantly smaller tables. We obtain an extension of L*, which we call CCLL* simply by replacing every command of the form $U := U \cup L$ (where L is an arbitrary set of words) by $U := U \cup \mathit{pnf}(L)$, and every command of the form $V := V \cup L$ by $V := V \cup \mathit{snf}(L)$.

The correctness of our improved algorithm is stated in the following theorem.

Theorem 5.0.1 (CCLL*: Correctness). *Let $\mathcal{D} \subseteq \Sigma^*$ be a domain and $\approx_{\mathcal{D}}$ be a congruence over $\mathcal{D}$ such that $\mathcal{D}$ satisfies* (*) – (***). *If the Teacher classifies/provides words in conformance with a regular language L, then invoking* CCLL* $(\mathcal{D}, \approx_{\mathcal{D}})$ *returns, after finitely many steps, an automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*

Proof: Consider an instance of (T, U, V) during a run of CCLL*. For $w \in (U \cup U\Sigma)V$, the value of $T(w)$ is $-$ if $w \notin \mathcal{D}$. If, on the other hand, $w \in \mathcal{D}$, then $T(w)$ only depends on the classification of w by the *Teacher*. So let $u, v \in \Sigma^*$. We consider the two abovementioned cases.

- Suppose $uv \notin \mathcal{D}$. Then, by (*), $u \notin \mathit{pref}(\mathcal{D})$ or $v \notin \mathit{suff}(\mathcal{D})$. Thus, $\mathit{pnf}(u) \notin \mathit{pref}(\mathcal{D})$ or $\mathit{snf}(v) \notin \mathit{suff}(\mathcal{D})$ so that finally $\mathit{pnf}(u) \cdot \mathit{snf}(v) \notin \mathcal{D}$.
- Suppose $uv \in \mathcal{D}$. Then, $u \in \mathit{pref}(\mathcal{D})$ and $v \in \mathit{suff}(\mathcal{D})$. By the definition of the mappings pnf and snf, there are u' and v' such that $\mathit{pnf}(u) \cdot v' \approx_{\mathcal{D}} uv'$ and $u' \cdot \mathit{snf}(v) \approx_{\mathcal{D}} u'v$. By (**) and (***), $\{\mathit{pnf}(u) \cdot v, u \cdot \mathit{snf}(v)\} \subseteq \mathcal{D}$ so that $\mathit{pnf}(u) \cdot v \approx_{\mathcal{D}} uv$ and $u \cdot \mathit{snf}(v) \approx_{\mathcal{D}} uv$. Applying (**) (or (***)) a second time, we obtain $\mathit{pnf}(u) \cdot \mathit{snf}(v) \in \mathcal{D}$. We deduce $\mathit{pnf}(u) \cdot \mathit{snf}(v) \approx_{\mathcal{D}} uv$, which implies $uv = \mathit{pnf}(u) \cdot \mathit{snf}(v)$.

Thus, it does not matter if an entry $T(w)$ in the table is made on the basis of w or on $\mathit{pnf}(u) \cdot \mathit{snf}(v)$, regardless of the partitioning uv of w. In particular, if we replace, in U and V, every word with its respective normal form, then the resulting table preserves consistency and closure properties. Moreover, the automaton that we can construct, given the new table is closed and consistent, is isomorphic to that of the original table.

As this replacement is precisely what is systematically done in CCLL*, the theorem follows. □

Table 5.1 highlights the changes that are necessary to integrate congruence-closed language learning into L* (or any similar table-based learning algorithm, e.g., $\mathrm{L}^*_{\mathrm{col}}$, NL*, or UL*). The update function CCL-T-UPDATE depicted in Table 5.2 is not changed compared to Table 3.5 on page 35 and only listed here for the sake of completeness.

Depending on the underlying automaton model, there are differences in how to obtain the final automaton from the derived table. We will now define the DFA for a table derived using the CCLL* algorithm. Note that for RFSA and RUFA this definition is very similar.

Definition 5.0.2 (Automaton of a table). *Given a closed and consistent table $\mathcal{T}_{\mathrm{CCLL}^*}$, the underlying hypothesis DFA is $\mathcal{H}_{\mathrm{CCLL}^*} = (Q, Q_0, \delta, F)$ where:*

- $Q = \{\mathit{row}(u) \mid u \in U\}$,

```
CCL L*($\mathcal{D}, \approx_{\mathcal{D}}$):
U := {ε}; V := {ε}; T is defined nowhere;
CCL-T-UPDATE();
repeat
        while ($T, U, V$) is not (closed and consistent)
            do
                if ($T, U, V$) is not consistent then
                    find $u, u' \in U, a \in \Sigma$, and $v \in V$ such that $row(u) = row(u')$ and
                                                         $row(ua)(v) \neq row(u'a)(v)$;
                    $V := V \cup snf(\{av\})$;
                    CCL-T-UPDATE();
                if ($T, U, V$) is not closed then
                    find $u \in U$ and $a \in \Sigma$ such that $row(ua) \neq row(u')$ for all $u' \in U$;
                    $U := U \cup pnf(\{ua\})$;
                    CCL-T-UPDATE();
        /* ($T, U, V$) is both closed and consistent, hence, $\mathcal{H}_{(T,U,V)}$ can be derived */
        perform equivalence test for $\mathcal{H}_{(T,U,V)}$;
        if equivalence test fails then
            get counterexample $w$;
            $U := U \cup pnf(pref(w))$;
            CCL-T-UPDATE();
    until equivalence test succeeds;
    return $\mathcal{H}_{(T,U,V)}$;
```

Table 5.1: CCL L*: Extension of L* by congruence-closed language learning

```
CCL-T-UPDATE ():
for $w \in (U \cup U\Sigma)V$ such that $T(w)$ is not defined
    $T(w) := getClassificationFromTeacher(w)$;
```

Table 5.2: Function for updating table function in CCL L*

- $Q_0 = \{row(\varepsilon)\}$,
- $\delta(row(u), a) = \{row(pnf(ua))\}$ *for* $row(u) \in Q$, $a \in \Sigma$, *and*
- $F = \{r \in Q \mid r(\varepsilon) = +\}$.

We would like to emphasize again that this approach is not only applicable to L* but to all learning algorithms (e.g., $\mathrm{L}^*_{\mathrm{col}}$, NL*, and UL*, with the slight difference that counterexamples and their suffixes are added to the set of columns) that implement Angluin's table. Moreover, in Sections 6.3 and 9.2 we will see how the learning of congruence-closed languages can be employed in the setting of message sequence charts to obtain significant reductions in memory consumption concerning Angluin's table.

In the following, we will give an example of how the normal forms are applied to tables and emphasize, for a given regular language, the gain obtained by this approach.

Example 5.0.3. Let $\Sigma = \{a, b\}$, $\mathcal{D} = \Sigma^*$, and consider the finite regular language $L = \{w \in \Sigma^* \mid |w|_a = |w|_b \leq 2\}$. Suppose furthermore that $\approx_{\mathcal{D}} = \{(w_1, w_2) \in \mathcal{D}^* \times \mathcal{D}^* \mid |w_1|_a = |w_2|_a$ and $|w_1|_b = |w_2|_b\}$. The normal forms in this example are $pnf(w) =$

$snf(w) = a^{|w|_a} b^{|w|_b}$ for any $w \in \Sigma^*$. It can easily be verified that this choice satisfies the requirements (*)–(***) that were imposed on the congruence relations and the normal forms. The goal is to infer the minimal automaton from Figure 5.1—which recognizes the regular language L—by means of L* and CCLL*, and to compare the resulting tables $\mathcal{T}_{L^*}$ and $\mathcal{T}_{\text{CCLL}^*}$ with respect to their sizes.

To get an impression of how the normalization of words using *pnf* and *snf* works, let us first and exemplarily consider a table containing the following two rows labeled by $p_1 = aba$ and $p_2 = baa$:

	ε	a	b	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$
$p_1 = aba$	$-$	$-$	$+$	$\cdots$
$p_2 = baa$	$-$	$-$	$+$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$\overset{pnf}{\Longrightarrow}$

	ε	a	b	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$
$pnf(p_1)$ $= aab$	$-$	$-$	$+$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The normal forms for p_1 and p_2 are $pnf(p_1) = pnf(p_2) = aab$. Therefore, these two rows are collapsed into one, as shown on the right hand side of the figure. Similarly, columns representing $\approx_{\mathcal{D}}$-congruent words can be collapsed. The following table contains suffixes s_1 and s_2. The normal forms of s_1 and s_2 result in the same suffix: $snf(s_1) = snf(s_2) = abb$.

	$\cdots$	$s_1 =$ bab	$s_2 =$ abb	$\cdots$
ε	$\cdots$	$-$	$-$	$\cdots$
a	$\cdots$	$+$	$+$	$\cdots$
b	$\cdots$	$-$	$-$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$\overset{snf}{\Longrightarrow}$

	$\cdots$	$snf(s_1)$ $= abb$	$\cdots$
ε	$\cdots$	$-$	$\cdots$
a	$\cdots$	$+$	$\cdots$
b	$\cdots$	$-$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Given a partially compressed or even uncompressed table, it might be interesting to fully collapse the rows of congruent words and use the resulting table for continuing the learning procedure. As collapsing operations only concern equivalent rows or columns, respectively, the order in which compression is deployed to a (partially) uncompressed table is not of importance. Thus, an uncompressed table can be reduced at any time or, as in the algorithm CCLL* from Tables 5.1 and 5.2, on-the-fly, during learning. ◇

Returning to the initial example, language L can now be inferred using the original L* algorithm and the new improved version CCLL*. The final tables are presented in Table 5.3. On the left hand, the resulting table $\mathcal{T}_{L^*}$ after applying L* is shown. For this rather small example, it already contains 275 entries. On the right side, table $\mathcal{T}_{\text{CCLL}^*}$ describes the same hypothesis automaton after applying the CCLL* algorithm, but the final table only contains 144 entries. This represents an improvement of approximately 47,7% regarding memory usage.

Of course, the usability of this modification substantially depends on the domain and the congruence that are employed. Thus, it has to be regarded as a domain-specific optimization which in some cases may cause fundamental improvements concerning memory consumption.

Note that the idea of exploiting an independence relation for learning is already appears in [EGPQ06] in the context of grey-box checking.

$\mathcal{T}_{\mathrm{L}^*}$	ε	b	a	bb	ab	ba	abb	$aabb$	aab	aba	aa
ε	+	−	−	−	+	+	−	+	−	−	−
a	−	+	−	−	−	−	+	−	−	−	−
aa	−	−	−	+	−	−	−	−	−	−	−
ab	+	−	−	−	+	+	−	−	−	−	−
ba	+	−	−	−	+	+	−	−	−	−	−
b	−	−	+	−	−	−	−	−	+	+	−
aaa	−	−	−	−	−	−	−	−	−	−	−
$aaaa$	−	−	−	−	−	−	−	−	−	−	−
$aabb$	+	−	−	−	−	−	−	−	−	−	−
aab	−	+	−	−	−	−	−	−	−	−	−
$bbaa$	+	−	−	−	−	−	−	−	−	−	−
bba	−	−	+	−	−	−	−	−	−	−	−
bb	−	−	−	−	−	−	−	−	−	−	+
aba	−	+	−	−	−	−	−	−	−	−	−
abb	−	−	+	−	−	−	−	−	−	−	−
baa	−	+	−	−	−	−	−	−	−	−	−
bab	−	−	+	−	−	−	−	−	−	−	−
$aaab$	−	−	−	−	−	−	−	−	−	−	−
$aaaaa$	−	−	−	−	−	−	−	−	−	−	−
$aaaab$	−	−	−	−	−	−	−	−	−	−	−
$aabba$	−	−	−	−	−	−	−	−	−	−	−
$aabbb$	−	−	−	−	−	−	−	−	−	−	−
$aaba$	−	−	−	−	−	−	−	−	−	−	−
$bbaaa$	−	−	−	−	−	−	−	−	−	−	−
$bbaab$	−	−	−	−	−	−	−	−	−	−	−

(a) uncompressed using L* (275 entries)

$\mathcal{T}_{\mathrm{CCLL}^*}$	ε	b	a	bb	ab	abb	$aabb$	aab	aa
ε	+	−	−	−	+	−	+	−	−
a	−	+	−	−	−	+	−	−	−
aa	−	−	−	+	−	−	−	−	−
ab	+	−	−	−	+	−	−	−	−
b	−	−	+	−	−	−	−	+	−
aaa	−	−	−	−	−	−	−	−	−
$aaaa$	−	−	−	−	−	−	−	−	−
$aabb$	+	−	−	−	−	−	−	−	−
aab	−	+	−	−	−	−	−	−	−
abb	−	−	+	−	−	−	−	−	−
bb	−	−	−	−	−	−	−	−	+
$aaab$	−	−	−	−	−	−	−	−	−
$aaaaa$	−	−	−	−	−	−	−	−	−
$aaaab$	−	−	−	−	−	−	−	−	−
$aaabb$	−	−	−	−	−	−	−	−	−
$aabbb$	−	−	−	−	−	−	−	−	−

(b) compressed using CCLL* (144 entries)

Table 5.3: Uncompressed and compressed table for regular language L from Example 5.0.3 using L* and CCLL*, respectively

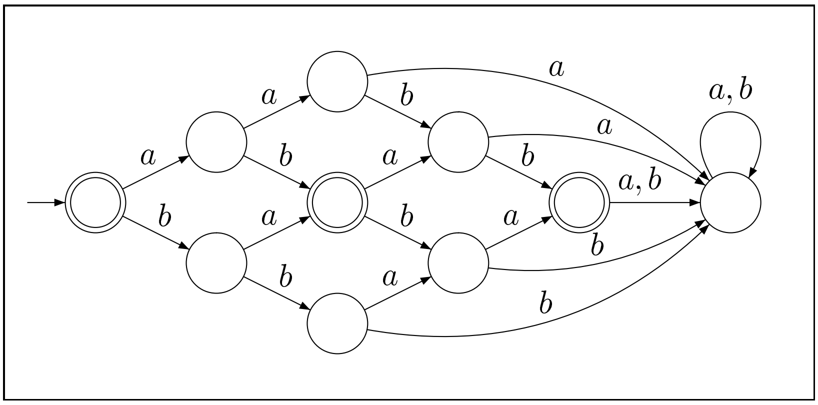


Figure 5.1: Minimal DFA for finite regular language L from Example 5.0.3

6 Learning and Synthesis of Distributed Systems

In the following chapter we will describe an application of the L* learning algorithm. The domain we are now dealing with is the domain of distributed systems, and the main goal we want to achieve is to detect classes of regular languages that represent distributed systems and that are learnable employing active online algorithms. The whole approach described here can be regarded as part of the software engineering development cycle as we will show in Chapter 7.

Software development usually starts with eliciting requirements. Requirement capturing techniques of various nature exist. Popular requirement engineering methods, such as the Inquiry Cycle and CREWS [NE00], exploit use cases and scenarios for specifying system's requirements. Scenarios given as sequence diagrams are also at the heart of the UML (Unified Modeling Language). A scenario is a partial fragment of the system's behavior given as visual representation indicating the system components (vertically, denoting time evolution) and their message exchange (horizontally). Their intuitive yet formal nature has led to a broad acceptance by the software engineering community, both in academia as well as in industry. Scenarios can be positive or negative, indicating either a possible desired or an unwanted system behavior, respectively. Different scenarios together form a more complete description of the system behavior.

Requirements capturing is typically followed by a first design step of the system at hand. This step naturally ignores many implementation details and aims to obtain an initial system structure at a high level of abstraction. In case of a distributed system realization this, e.g., amounts to determine which processes are to be distinguished, what their high-level behavior is, and which capacities of communication channels suffice to warrant a deadlock-free process interaction. This design phase in software engineering is highly challenging as it concerns a complex paradigm shift between the requirement specification—a partial, overlapping and possibly inconsistent description of the system's behavior that is subject to rapid change—and a conforming design model, a first complete behavioral description of the system.

The fact that target systems are often distributed complicates matters considerably as combining several individual processes may easily yield realizations that handle more than the specified scenarios, i.e., they may over-approximate the system requirements, or suffer from deadlocks. During the synthesis of such distributed design models, conflicting requirements are detected and resolved. As a consequence, the requirements specification is adapted by adding or omitting scenarios. Besides, a thorough analysis of the design model, e.g., by means of model checking or simulation, requires fixing errors in the requirements. Obtaining a complete and consistent set of requirements together with a conforming design model is thus a complex and highly iterative process.

In the rest of this chapter we consider requirements given as message sequence charts (MSCs). MSCs are a standardized notation [ITU96, ITU99, ITU04] for modeling behavior of distributed systems. The MSCs regarded in this thesis are basic; high-level constructs to combine MSCs by alternative, sequential or repetitive composition are not considered. This yields a simple, yet still effective requirement specification formalism that is expressive, and easy to grasp and to understand. For the design models we focus on distributed systems where each process behavior is described as a finite-state machine and processes exchange messages asynchronously via order-preserving communication channels. These communicating finite-state machines (CFMs [BZ83]) are commonly adopted for realizing MSCs [AEY01, BM07, Gen05, GKM06, GKM07, GMSZ06, HMK$^+$05, Loh03, Mor02].

The goal now consists of exploiting Angluin's learning algorithm (cf. Subsection 3.3.1) to synthesize CFMs from requirements given as sets of (positive and negative) basic MSCs. Learning fits well with the incremental generation of design models as it is feasible to infer a design model on the basis of an initial set of scenarios, CFMs are adapted in an automated manner on adding and deletion of MSCs, and diagnostic feedback is provided that may guide an amendment of the requirements when establishing an inconsistency of a set of scenarios. The use of learning for system synthesis from scenario-based requirements specifications is not new and has been proposed by several authors, see, e.g., [MS01, UBC07, UKM03, DLD05]. The main characteristics of our approach are the unique combination of the following aspects:

(i) Positive and *negative* MSCs are naturally supported.

(ii) The realized processes interact *fully asynchronously*.

(iii) The synthesized CFMs *precisely* exhibit the behavior as specified by the input MSCs.

(iv) Some effective optimizations tailored to *partial orders* (e.g., MSCs) are developed.

Existing learning-based synthesis techniques typically consider just possible and no undesired behaviors, yield synchronously (or partially asynchronously) interacting automata, and, most importantly, suffer from the fact that synthesized realizations may exhibit more behavior than specified so that the obtained realizations are in fact over-approximations. For our approach, however, it is guaranteed that the final system will precisely exhibit the behavior that was presented to the approach, which means that all scenarios specified as positive will be included in the target language and all negatively specified behavior excluded from the language.

6.1 Preliminaries

In this section, we introduce two fundamental concepts: *message sequence charts* (MSCs) and *communicating finite-state machines* (CFMs) [BZ83]. The former constitute an appealing and easy to understand graphical specification formalism. The latter, also known as *message passing automata* (MPA), serve as design models for the system to learn and model the communication behavior of distributed systems composed of finite-state components.

As in this chapter we are dealing with communicating entities, which are capable of executing sending and receiving actions, elements of communication alphabets are called *actions*.

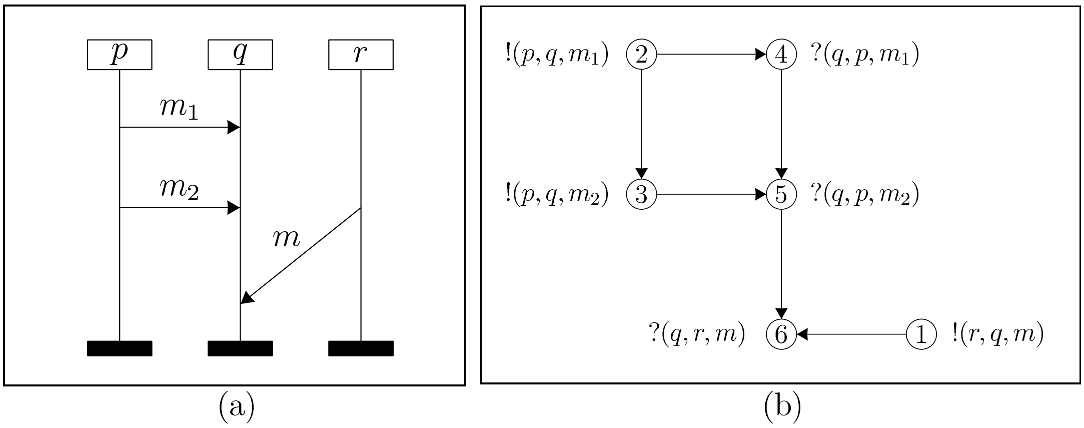
6.1.1 Message Sequence Charts

A common design practice when developing communicating systems is to start with specifying scenarios to exemplify the intended interaction of the system to be. *Message sequence charts* (MSCs) provide a prominent notion to further this approach. They are widely used in industry, are standardized [ITU98, ITU04], and resemble UML's sequence diagrams [Ara98]. An MSC depicts a single partially ordered execution sequence of a system. It consists of a collection of processes, which, in their visual representation, are drawn as vertical lines and are interpreted as top-down time axes. Moreover, an arrow from one line to a second corresponds to the communication events of sending and receiving a message. An example MSC is illustrated in Figure 6.1(a). The benefit of such a diagram is that one grasps its meaning at a glance. In the example scenario, messages m_1 and m_2 are sent from process p to process q. A further message m originates at process r and is finally received at q. However, one still has to reach an agreement on the system architecture, which does not necessarily emerge from the picture. Namely, following the MSC standard, we assume asynchronous communication: the send and receipt of a message might happen time-delayed. More precisely, there is an unbounded FIFO channel in between two processes that allows a sender process to proceed while the message is waiting for being received. Moreover, we assume a single process to be sequential: the events of one particular process are totally ordered in accordance with their appearance on its time axis. For example, regarding Figure 6.1(a), we suppose that sending m_2 occurs after sending m_1. However, as the relative speed of the processes is unknown, we do not know if m_1 is received before m_2 is sent. Thus, the latter two events remain unordered.

We conclude that, in a natural manner, an MSC can be seen as a labeled partial order (labeled poset) over its events. Figure 6.1(b) depicts the Hasse diagram of the labeled poset that one would associate with the diagram from Figure 6.1(a). Its elements $1, \ldots, 6$ represent the endpoints of the message arrows and are called *events*. The edge relation then reflects the two constraints on the order of execution of the events: (i) events that are located on the same process line are totally ordered, and (ii) a send event has to precede the corresponding receive event. Indeed, it is reasonable to require that the transitive closure of constraints (i) and (ii) is a partial order. To keep track of the nature of an event in the poset representation, any such event is labeled with an action. Thus, a possible label is either:

- a send action, which is of the form $!(p, q, m)$ meaning that p sends a message m to q or
- a receive action, which is of the form $?(q, p, m)$ and is the complementary receive action executed by process q.

The alphabet of actions is, therefore, parameterized by nonempty and finite sets *Proc* of *processes* and *Msg* of *messages*, which we suppose to be fixed in the following. We suppose $|\mathit{Proc}| \geq 2$. Recall that we assume an exchange of messages through channels. The set of *channels* is denoted $\mathit{Ch} = \{(p, q) \in \mathit{Proc} \times \mathit{Proc} \mid p \neq q\}$. The set Act_p of *actions* that may be executed by process p is given by $\mathit{Act}_p = \{!(p, q, m) \mid (p, q) \in \mathit{Ch}$ and $m \in \mathit{Msg}\} \cup \{?(p, q, m) \mid (p, q) \in \mathit{Ch}$ and $m \in \mathit{Msg}\}$. Moreover, let $\mathit{Act} = \bigcup_{p \in \mathit{Proc}} \mathit{Act}_p$ denote the set of all actions. Before we formally define what we understand by an MSC, let us first consider general *Act*-labeled posets, i.e., structures $(E, \preceq, \lambda)$ where E is a finite set of *events*, λ is a labeling function of the form $E \to \mathit{Act}$, and $\preceq$ is a partial-order relation (it is reflexive, transitive, and antisymmetric). For process $p \in \mathit{Proc}$, let



(a) (b)

Figure 6.1: An MSC as a diagram (a) and as a graph (b)

$\preceq_p := \preceq \cap (E_p \times E_p)$ be the restriction of $\preceq$ to $E_p := \lambda^{-1}(Act_p)$ (which will later be required to give rise to a total order) and the direct predecessor relation on process p: $\lessdot_p := \{(e,e') \in \preceq_p \mid \nexists e'' \in E_p : e \preceq_p e'' \preceq_p e' \text{ and } e \neq e'' \neq e'\}$. Moreover, we define the relation $\prec_{\text{msg}} \subseteq E \times E$ to detect corresponding send and receive events: $i \prec_{\text{msg}} j$ if there are a channel $(p,q) \in Ch$ and a message $m \in Msg$ such that:

- $\lambda(i) = !(p,q,m)$, $\lambda(j) = ?(q,p,m)$ and
- $|\{i' \preceq i \mid \lambda(i') = !(p,q,m') \text{ for some } m' \in Msg\}| =$
  $|\{j' \preceq j \mid \lambda(j') = ?(q,p,m') \text{ for some } m' \in Msg\}|$.

That is, events i and j correspond to a message exchange only if the number of messages that have been sent through channel (p,q) before i equals the number of messages that have been received before j. This ensures FIFO communication.

Definition 6.1.1 (Message Sequence Chart (MSC)). *An* MSC *is an Act-labeled poset* $(E, \preceq, \lambda)$ *such that:*

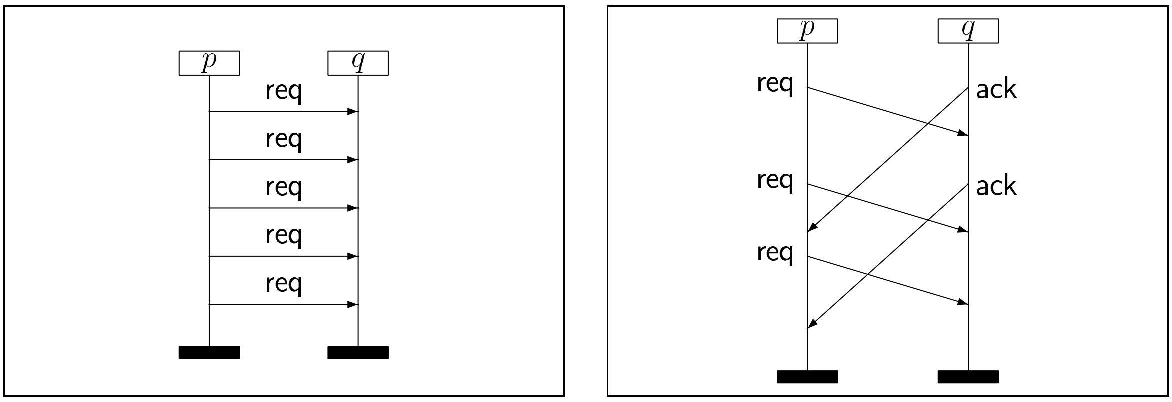
- *for all* $p \in Proc$, $\preceq_p$ *is a total order on* E_p,
- $\preceq = (\prec_{\text{msg}} \cup \bigcup_{p \in Proc} \preceq_p)^*$, *and*
- *every event is part of a message, i.e., for every* $i \in E$, *there is* $j \in E$ *such that either* $(i,j) \in \prec_{\text{msg}}$ *or* $(j,i) \in \prec_{\text{msg}}$.

See Figure 6.1(a) and Figure 6.2 for graphical representations of some example MSCs.

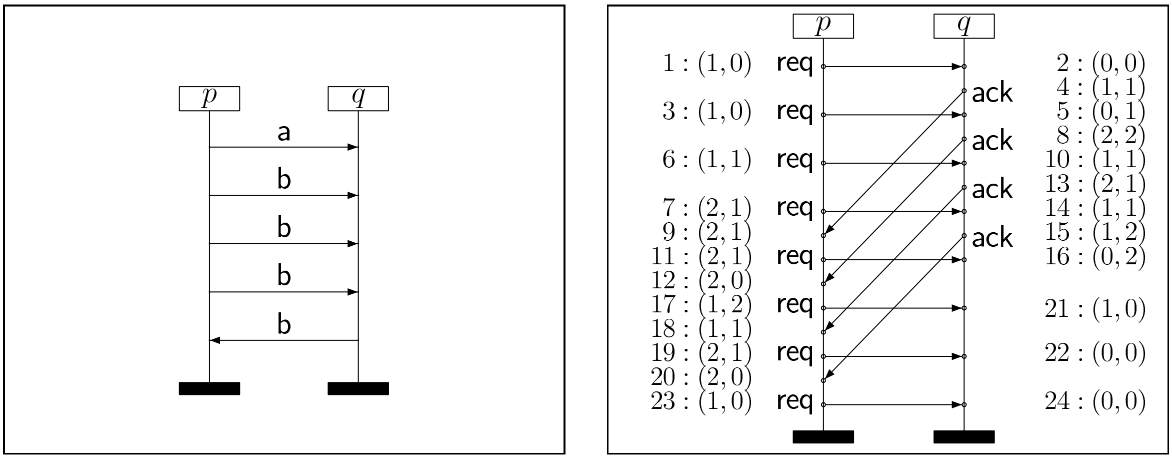
Stated in words, an MSC is an *Act*-labeled poset such that events occurring at a single process are totally ordered, and that for each send event i there is a corresponding receive event j with $i \prec_{\text{msg}} j$. For these events the order is fixed. Independent events, though, can occur in any order.

Sequential observations of labeled posets are called linearizations. A *linearization* of an *Act*-labeled poset $(E, \preceq, \lambda)$ is any saturation of $\preceq$ to a total order $\preceq'$, i.e., $e_{i_1} \preceq' \ldots \preceq' e_{i_n}$, where $(i_1, \ldots, i_n)$ is a permutation of $(1, \ldots, n)$ such that, for all $j, k \in \{1, \ldots, n\}$, $e_{i_j} \preceq e_{i_k}$ implies $j \leq k$. A linearization $e_{i_1} \ldots e_{i_n}$ corresponds to the word $\lambda(e_{i_1}) \ldots \lambda(e_{i_n}) \in Act^*$ and, by abuse of nomenclature, we call $\lambda(e_{i_1}) \ldots \lambda(e_{i_n})$ a linearization as well. For example,

$$!(r,q,m)\,!(p,q,m_1)\,!(p,q,m_2)\,?(q,p,m_1)\,?(q,p,m_2)\,?(q,r,m)$$


p
q
req
req
req
req
req
p
q
req
ack
req
ack
req

(a) MSC M_{a} (b) MSC M_{b}


p
q
a
b
b
b
b
p
q
1 : (1, 0) req
3 : (1, 0) req
6 : (1, 1) req
7 : (2, 1) req
9 : (2, 1)
11 : (2, 1) req
12 : (2, 0)
17 : (1, 2) req
18 : (1, 1)
19 : (2, 1) req
20 : (2, 0)
23 : (1, 0) req
ack
ack
ack
ack
2 : (0, 0)
4 : (1, 1)
5 : (0, 1)
8 : (2, 2)
10 : (1, 1)
13 : (2, 1)
14 : (1, 1)
15 : (1, 2)
16 : (0, 2)
21 : (1, 0)
22 : (0, 0)
24 : (0, 0)

(c) MSC M_{c} (d) MSC M_{d}

Figure 6.2: A collection of message sequence charts

is a linearization of the MSC in Figure 6.1(a). The set of linearizations of a labeled poset $\mathscr{M}$ will be denoted by $\mathit{Lin}(\mathscr{M})$. This mapping is canonically extended towards sets $\mathcal{L}$ of partial orders: $\mathit{Lin}(\mathcal{L}) = \bigcup_{\mathscr{M} \in \mathcal{L}} \mathit{Lin}(\mathscr{M})$.

6.1.2 Communicating Finite-State Machines

MSCs constitute a visual high-level specification formalism. They can be represented graphically and offer an intuitive semantics (in terms of their linearizations). On the computational side, we consider automata models that reflect the kind of communication that is depicted in an MSC. We now turn towards an automata model that, in a natural manner, generates collections of MSCs. More precisely, it generates action sequences that follow an *all-or-none* law: either all linearizations of an MSC are generated, or none of them.

A communicating finite-state machine (CFM) is a collection of finite-state machines, one for each process. According to the assumptions that we made for MSCs, we assume that communication between these machines takes place via (a priori) unbounded reliable FIFO channels. The underlying system architecture is again parameterized by the set *Proc* of processes and the set *Msg* of messages. Recall that this gives rise to the set *Act* of

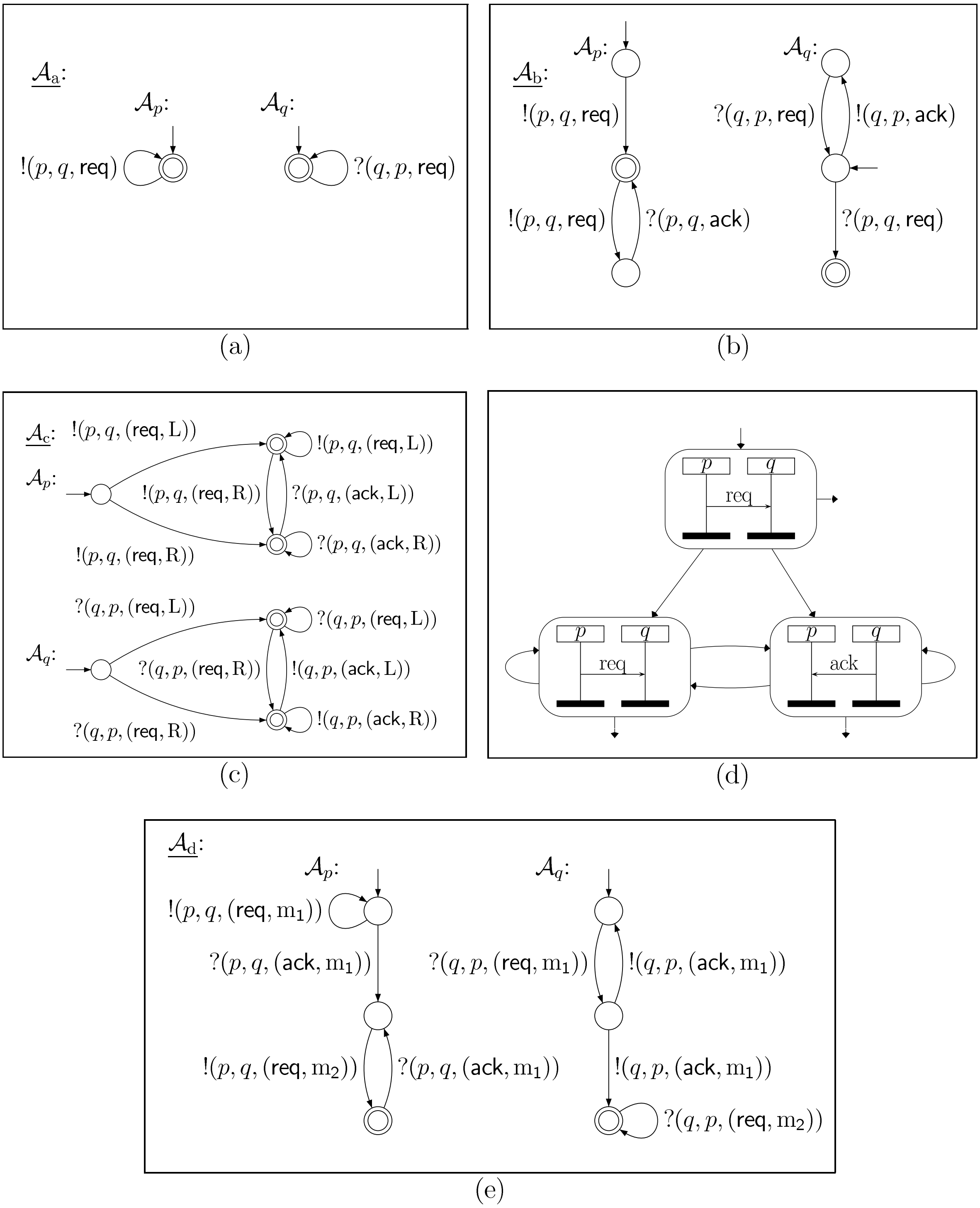

$\underline{\mathcal{A}_{\mathrm{a}}}$:
$\mathcal{A}_p$:
$\mathcal{A}_q$:
$!(p, q, \mathsf{req})$
$?(q, p, \mathsf{req})$
(a)
$\underline{\mathcal{A}_{\mathrm{b}}}$:
$\mathcal{A}_p$:
$\mathcal{A}_q$:
$!(p, q, \mathsf{req})$
$!(p, q, \mathsf{req})$
$?(p, q, \mathsf{ack})$
$?(q, p, \mathsf{req})$
$!(q, p, \mathsf{ack})$
$?(p, q, \mathsf{req})$
(b)
$\underline{\mathcal{A}_{\mathrm{c}}}$:
$\mathcal{A}_p$:
$!(p, q, (\mathsf{req}, \mathrm{L}))$
$!(p, q, (\mathsf{req}, \mathrm{L}))$
$!(p, q, (\mathsf{req}, \mathrm{R}))$
$?(p, q, (\mathsf{ack}, \mathrm{L}))$
$?(p, q, (\mathsf{ack}, \mathrm{R}))$
$!(p, q, (\mathsf{req}, \mathrm{R}))$
$\mathcal{A}_q$:
$?(q, p, (\mathsf{req}, \mathrm{L}))$
$?(q, p, (\mathsf{req}, \mathrm{L}))$
$?(q, p, (\mathsf{req}, \mathrm{R}))$
$!(q, p, (\mathsf{ack}, \mathrm{L}))$
$!(q, p, (\mathsf{ack}, \mathrm{R}))$
$?(q, p, (\mathsf{req}, \mathrm{R}))$
(c)
p
q
req
p
q
req
p
q
ack
(d)
$\underline{\mathcal{A}_{\mathrm{d}}}$:
$\mathcal{A}_p$:
$\mathcal{A}_q$:
$!(p, q, (\mathsf{req}, \mathrm{m}_1))$
$?(p, q, (\mathsf{ack}, \mathrm{m}_1))$
$!(p, q, (\mathsf{req}, \mathrm{m}_2))$
$?(p, q, (\mathsf{ack}, \mathrm{m}_1))$
$?(q, p, (\mathsf{req}, \mathrm{m}_1))$
$!(q, p, (\mathsf{ack}, \mathrm{m}_1))$
$!(q, p, (\mathsf{ack}, \mathrm{m}_1))$
$?(q, p, (\mathsf{req}, \mathrm{m}_2))$
(e)

Figure 6.3: A collection of communicating finite-state machines

actions, which will provide the transition labelings. In our automata model, the effect of executing a send action of the form $!(p,q,m)$ by process p is to put message m at the end of the channel (p,q) from process p to process q. Receive actions, written as $?(q,p,m)$, are only enabled when the requested message m is found at the head of the channel (p,q). When enabled, its execution by process q removes the corresponding message m from the channel from p to q.

It has been shown that the CFM model derived from concepts explained so far has a limited expressiveness. Certain protocols cannot be implemented without possible deadlocks by CFMs, unless the model is enriched by so-called *control-* or *synchronization messages* [BM03, BL05]. Intuitively, we cannot guarantee to get a deadlock-free CFM if we have too few synchronization messages. If, e.g., one process has got $n+1$ states but we are only allowed to use n synchronization messages a different process cannot definitely be sure in which state the other process is currently in. Therefore, we extend our alphabet wrt. a fixed infinite supply of control messages Λ. Let Act_p^{Λ} contain the symbols of the form $!(p,q,(m,\lambda))$ or $?(p,q,(m,\lambda))$ where $!(p,q,m) \in Act_p$ (respectively $?(p,q,m) \in Act_p$) and $\lambda \in \Lambda$. Intuitively, we tag messages with some control information λ to circumvent deadlocks. Finally, let $Act^{\Lambda} = \bigcup_{p \in Proc} Act_p^{\Lambda}$.

Definition 6.1.2 (Communicating Finite-State Machine (CFM))**.** *A* communicating finite-state machine *(CFM) is a structure $\mathcal{A} = ((\mathcal{A}_p)_{p \in Proc}, \mathcal{I})$. For any process $p \in Proc$, $\mathcal{A}_p = (S_p, \Delta_p, F_p)$ constitutes the behavior of p where:*

- *S_p is a finite set of* (local) states*,*
- *$\Delta_p \subseteq S_p \times Act_p^{\Lambda} \times S_p$ is the finite* transition relation*, and*
- *$F_p \subseteq S_p$ is the set of* final states*.*

Moreover, $\mathcal{I} \subseteq \prod_{p \in Proc} S_p$ is the set of global initial states.

For an example CFM, consider Figure 6.3(c) where $Proc = \{p,q\}$, $Msg = \{\mathsf{req}, \mathsf{ack}\}$, and $\{\mathrm{L}, \mathrm{R}\} \subset \Lambda$, or Figure 6.3(e) where $Proc = \{p,q\}$, $Msg = \{\mathsf{req}, \mathsf{ack}\}$, and $\{\mathrm{m}_1, \mathrm{m}_2\} \subset \Lambda$.

Let $\mathcal{A} = ((\mathcal{A}_p)_{p \in Proc}, \mathcal{I})$ with $\mathcal{A}_p = (S_p, \Delta_p, F_p)$ be a CFM. The *size of* $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined to be $\sum_{p \in Proc} |S_p|$. A *configuration* of $\mathcal{A}$ gives a snapshot of the current state of each process and the current channel contents. Thus, the set of configurations of $\mathcal{A}$, denoted by $Conf_{\mathcal{A}}$, consists of pairs $(\overline{s}, \chi)$ with $\overline{s} \in \prod_{p \in Proc} S_p$ a global state and $\chi : Ch \to (Msg \times \Lambda)^*$, determining the channel contents. The set of *initial configurations* is defined as $\mathcal{I} \times \{\chi_\varepsilon\}$ where χ_ε maps each channel onto the empty word, representing an empty channel. Analogously, the set of *final configurations* is $\prod_{p \in Proc} F_p \times \{\chi_\varepsilon\}$, i.e., each component is in a local final state and all messages are received at last. The projection of a global state $\overline{s} \in \prod_{p \in Proc} S_p$ to process p is denoted by $\overline{s}_p$. An execution of a send or receive action transfers the CFM from one configuration to another, according to the *global transition relation* of $\mathcal{A}$. This transition relation $\Longrightarrow_{\mathcal{A}} \subseteq Conf_{\mathcal{A}} \times Act \times Conf_{\mathcal{A}}$ is given by the following two inference rules. The first rule considers the sending of a message m from p to q and is given by:

$$\frac{(\overline{s}_p, !(p,q,(m,\lambda)), \overline{s}'_p) \in \Delta_p \text{ and for all } r \neq p, \overline{s}_r = \overline{s}'_r}{((\overline{s}, \chi), !(p,q,m), (\overline{s}', \chi')) \in \Longrightarrow_{\mathcal{A}}}$$

where $\chi' = \chi[(p,q) := (m,\lambda) \cdot \chi((p,q))]$, i.e., χ' maps (p,q) to the concatenation of (m,λ) and $\chi((p,q))$; for all other channels, χ' coincides with χ. This rule expresses that if the local automaton $\mathcal{A}_p$ has a transition labeled by $!(p, q, (m, \lambda))$ moving from state $\overline{s}_p$ to $\overline{s}'_p$, then the CFM $\mathcal{A}$ has a transition from $\overline{s}$ to $\overline{s}'$ where only the p component of $\mathcal{A}$ changes its state, and the new message (m, λ) is appended to the end of channel (p, q). Note that the control message has been abstracted away from the action that has been encountered when taking the transition. In other words, the transition is labeled by an element from *Act*, which follows our intuition that elements of Λ are only used for synchronization but do not contribute to observable behavior.

The second rule is complementary and considers the receipt of a message:

$$\frac{(\overline{s}_p, ?(p, q, (m, \lambda)), \overline{s}'_p) \in \Delta_p \text{ and for all } r \neq p, \overline{s}_r = \overline{s}'_r}{((\overline{s}, \chi), ?(p, q, m), (\overline{s}', \chi')) \in \Longrightarrow_{\mathcal{A}}}$$

where $\chi((q, p)) = w \cdot (m, \lambda) \neq \varepsilon$ and $\chi' = \chi[(q, p) := w]$. This rule states that if the local automaton $\mathcal{A}_p$ has a transition labeled by $?(p, q, (m, \lambda))$ moving from state $\overline{s}_p$ to $\overline{s}'_p$ then the CFM $\mathcal{A}$ has a transition from $\overline{s}$ to $\overline{s}'$, which is labeled with $?(p, q, m)$, where only the p component of $\mathcal{A}$ changes its state and the message (m, λ) is removed from the head of channel (q, p).

A *run* of CFM $\mathcal{A}$ on a word $w = a_1 \dots a_n \in Act^*$ is a sequence $c_0 \dots c_n \in Conf^*_{\mathcal{A}}$ of configurations where c_0 is an initial configuration and, for every $i \in \{1, \dots, n\}$, $(c_{i-1}, a_i, c_i) \in \Longrightarrow_{\mathcal{A}}$. The run is *accepting* if $c_n \in (\prod_{p \in Proc} F_p) \times \{\chi_\varepsilon\}$, i.e., each process is in an accepting state and all messages have been received yielding empty channels. Here, we first define the word semantics of CFMs, but later we will also consider their MSC semantics. Hence, for now, the *language* of a CFM $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words $w \in Act^*$ such that there is an accepting run of $\mathcal{A}$ on w.

Closure Properties of CFM Languages

To classify a word as either being a linearization of an MSC or not, we define the notion of *proper* and *well-formed* words.

Definition 6.1.3 (Proper and Well-formed Words). *We call* $w = a_1 \dots a_n \in Act^*$ *with* $a_i \in Act$ proper *if*

- *every receive action in* w *is preceded by a corresponding send action, i.e., for each channel* $(p, q) \in Ch$*, message* $m \in Msg$*, and prefix* u *of* w*, we have* $\sum_{m \in Msg} |u|_{!(p,q,m)} \geq \sum_{m \in Msg} |u|_{?(q,p,m)}$ *where* $|u|_a$ *denotes the number of occurrences of action* a *in the word* u*, and*

- *the FIFO policy is respected, i.e., for all* $1 \leq i < j \leq n$*,* $(p, q) \in Ch$*, and* $m_1, m_2 \in Msg$ *with* $a_i = !(p, q, m_1)$*,* $a_j = ?(q, p, m_2)$*, and* $|\{i' \leq i \mid a_{i'} = !(p, q, m) \text{ for some } m \in Msg\}| = |\{j' \leq j \mid a_{j'} = ?(q, p, m) \text{ for some } m \in Msg\}|$*, we have* $m_1 = m_2$*.*

A proper word w *is called* well-formed *if it satisfies* $\sum_{m \in Msg} |w|_{!(p,q,m)} = \sum_{m \in Msg} |w|_{?(q,p,m)}$.

Obviously, a run of a CFM on a word w only exists if w is proper, as a receive action is only enabled if the corresponding send message is at the head of the channel. Moreover, every word accepted by a CFM is well-formed, as acceptance implies empty channels.

In addition, as different processes interact asynchronously and, in general, independently, the language of a CFM is closed under a certain permutation rewriting. For example, consider a run of a CFM on the well-formed word:

$$!(p, q, m_1)\, !(p, q, m_2)\, ?(q, p, m_1)\, ?(q, p, m_2)\, !(r, q, m)\, ?(q, r, m),$$

i.e., process p sends a message m_1 to process q, followed by a message m_2, whereupon process q receives these messages in the correct order. We observe that process q could have received the message m_1 before sending m_2. Indeed, any CFM accepting the above action sequence will also accept the word:

$$!(p,q,m_1)\,?(q,p,m_1)\,!(p,q,m_2)\,?(q,p,m_2)\,!(r,q,m)\,?(q,r,m),$$

where any message is immediately received. Moreover, the action $!(r,q,m)$ is completely independent of all the actions that are engaged in sending/receiving the messages m_1 or m_2. Thus, a CFM cannot distinguish between the above sequences and sequence:

$$!(r,q,m)\,!(p,q,m_1)\,?(q,p,m_1)\,!(p,q,m_2)\,?(q,p,m_2)\,?(q,r,m).$$

Actually, $!(r,q,m)$ can be placed at any arbitrary position with the restriction that it has to occur before the complementary receipt of m. Note that the three well-formed words mentioned above all correspond to linearizations of the MSC from Figure 6.1(a) on page 84.

To capture the closure properties of a CFM formally, we identify labeled posets whose linearizations satisfy the *all-or-none* law, stating that either every or no linearization is accepted by a CFM. To this aim, we associate to any word $w = a_1 \ldots a_n \in \mathit{Act}^*$ an Act-labeled poset $\mathscr{M}(w) = (E, \preceq, \lambda)$ such that w is a linearization of $\mathscr{M}(w)$, and a CFM cannot distinguish between w and all other linearizations of $\mathscr{M}(w)$. The set of events is given by the set of positions in w, i.e., $E = \{1, \ldots, n\}$. Naturally, any position $i \in E$ is labeled with a_i, i.e., $\lambda(i) = a_i$. It remains to fix the partial-order relation $\preceq$, which reflects the dependencies between events. Clearly, we consider those events to be dependent that are executed by the same process or constitute the send and receipt of a message, since each process acts sequentially, and a message has to be sent before it can be received. Hence, let $\preceq_p \subseteq E_p \times E_p$ with E_p as before be defined by $i \preceq_p j$ iff $i \leq j$. Moreover, let $i \prec_{\mathrm{msg}} j$ if there is a channel $(p,q) \in \mathit{Ch}$ and a message $m \in \mathit{Msg}$ such that:

- $\lambda(i) = \,!(p,q,m)$, $\lambda(j) = \,?(q,p,m)$ and
- $|\{i' \leq i \mid \lambda(i') = \,!(p,q,m') \text{ for some } m' \in \mathit{Msg}\}| =$
  $|\{j' \leq j \mid \lambda(j') = \,?(q,p,m') \text{ for some } m' \in \mathit{Msg}\}|$.

This is similar to the definition of $\prec_{\mathrm{msg}}$ in the previous paragraph. Let $\preceq \,=\, (\prec_{\mathrm{msg}} \cup \bigcup_{p \in \mathit{Proc}} \preceq_p)^*$. To exemplify these notions, consider the well-formed word w defined as $!(r,q,m)\,!(p,q,m_1)\,!(p,q,m_2)\,?(q,p,m_1)\,?(q,p,m_2)\,?(q,r,m)$. Figure 6.1(b) depicts the Hasse diagram of the Act-labeled poset $\mathscr{M}(w) = (E, \preceq, \lambda)$. Note that $\mathscr{M}(w)$ is an MSC. Indeed, we have the following two lemmas, which are considered standard in the MSC literature (see, for example, [HMK+05]).

Lemma 6.1.4. *For any MSC $\mathscr{M}$, $w \in \mathit{Lin}(\mathscr{M})$ is well-formed.*

Lemma 6.1.5. *For any well-formed $w \in \mathit{Act}^*$, $\mathscr{M}(w)$ is an MSC. Moreover, $\mathscr{M}(w)$ and $\mathscr{M}(w')$ are isomorphic for all $w' \in \mathit{Lin}(\mathscr{M}(w))$.*

These results suggest to introduce an equivalence relation over well-formed words. The well-formed words w and w' are *equivalent*, written $w \approx w'$, if $\mathscr{M}(w)$ and $\mathscr{M}(w')$ are isomorphic. Note that this holds iff $w \in \mathit{Lin}(\mathscr{M}(w'))$.

Lemma 6.1.6 ([AEY01])**.** *For any CFM $\mathcal{A}$:*

(i) $L(\mathcal{A})$ *consists of well-formed words only, and*

(ii) $L(\mathcal{A})$ *is closed under* $\approx$.

The last claim asserts that for all well-formed words u and v with $u \approx v$, we have $u \in L(\mathcal{A})$ iff $v \in L(\mathcal{A})$. For a well-formed word w, let $[w]_\approx$ be the set of well-formed words that are equivalent to w wrt. $\approx$. For a set L of well-formed words, let $[L]_\approx := \bigcup_{w \in L} [w]_\approx$ be the *closure* of L wrt. $\approx$. The fact that $L(\mathcal{A})$ is closed under $\approx$, allows us to assign to $\mathcal{A}$ its set of MSCs $\mathcal{L}(\mathcal{A}) := \{\mathscr{M}(w) \mid w \in L(\mathcal{A})\}$. (Here, we identify isomorphic structures, i.e., we consider isomorphism classes of MSCs). This is an equivalent, visual, and more compact description of the behavior of CFM $\mathcal{A}$. Observe that $\mathit{Lin}(\mathcal{L}(\mathcal{A})) = L(\mathcal{A})$, i.e., the linearizations of the MSCs of CFM $\mathcal{A}$ correspond to its word language.

6.1.3 Deadlock-Free, Bounded, and Weak CFMs

In distributed computations, the notions of determinism, deadlock, and bounded channels play an important role [GMSZ06, GKM07, BM03]. Roughly speaking, a CFM is deterministic if every possible execution allows for at most one run; it is deadlock-free if any run can be extended towards an accepting one.

Definition 6.1.7 (Deterministic CFM). *A CFM* $\mathcal{A} = ((\mathcal{A}_p)_{p \in \mathit{Proc}}, \mathcal{I})$ *with* $\mathcal{A}_p = (S_p, \Delta_p, F_p)$ *is* deterministic *if, for all* $p \in \mathit{Proc}$, Δ_p *satisfies the following two conditions: (i) If we have both* $(s, !(p, q, (m, \lambda_1)), s_1) \in \Delta_p$ *and* $(s, !(p, q, (m, \lambda_2)), s_2) \in \Delta_p$, *then* $\lambda_1 = \lambda_2$ *and* $s_1 = s_2$. *(ii) If we have both* $(s, ?(p, q, (m, \lambda)), s_1) \in \Delta_p$ *and* $(s, ?(p, q, (m, \lambda)), s_2) \in \Delta_p$, *then* $s_1 = s_2$.

The CFMs from Figure 6.3(a) and (b) are deterministic whereas the CFMs from Figure 6.3(c) and (e) are not.

Definition 6.1.8 (Deadlock-free CFM ([GMSZ06])). *A CFM* $\mathcal{A}$ *is* deadlock-free *if, for all* $w \in \mathit{Act}^*$ *and all runs* γ *of* $\mathcal{A}$ *on* w, *there exist* $w' \in \mathit{Act}^*$ *and* $\gamma' \in \mathit{Conf}_{\mathcal{A}}^*$ *such that* $\gamma\gamma'$ *is an accepting run of* $\mathcal{A}$ *on* ww'.

The CFMs from Figure 6.3(a), (b), and (c) are deadlock-free. Note that, however, the CFM in Figure 6.3(c) will contain a deadlock if the control messages L and R were omitted. The CFM from Figure 6.3(e) contains a deadlock, even though synchronization messages are used. Imagine a run $\sigma = !(p, q, (\mathsf{req}, \mathrm{m}_1))$, $?(q, p, (\mathsf{req}, \mathrm{m}_1))$, $!(q, p, (\mathsf{ack}, \mathrm{m}_1))$, $?(p, q, (\mathsf{ack}, \mathrm{m}_1))$, $!(p, q, (\mathsf{req}, \mathrm{m}_2))$. The deadlock configuration resulting from σ is depicted in Figure 6.4. The local machine $\mathcal{A}_p$ is in its lowest state p_3 which is a final state. As the input buffers of this process are empty, the acceptance condition for this process is fulfilled. At the same time, machine $\mathcal{A}_q$ is in the non-accepting local state q_1 and is required to receive the action $(\mathsf{req}, \mathrm{m}_2)$ from buffer $\chi((p, q))$. The only possible action process q is able to perform, however, is action $(\mathsf{req}, \mathrm{m}_1)$. Hence, the CFM deadlocks. Note that removing the synchronization messages from the CFM does not resolve this problem, because then a different language will be recognized by the resulting CFM.

We obtain another essential restriction of CFMs if we require that any channel has a bounded capacity, say, $B \in \mathbb{N}$. Towards this notion, we first define when a word is B-bounded.

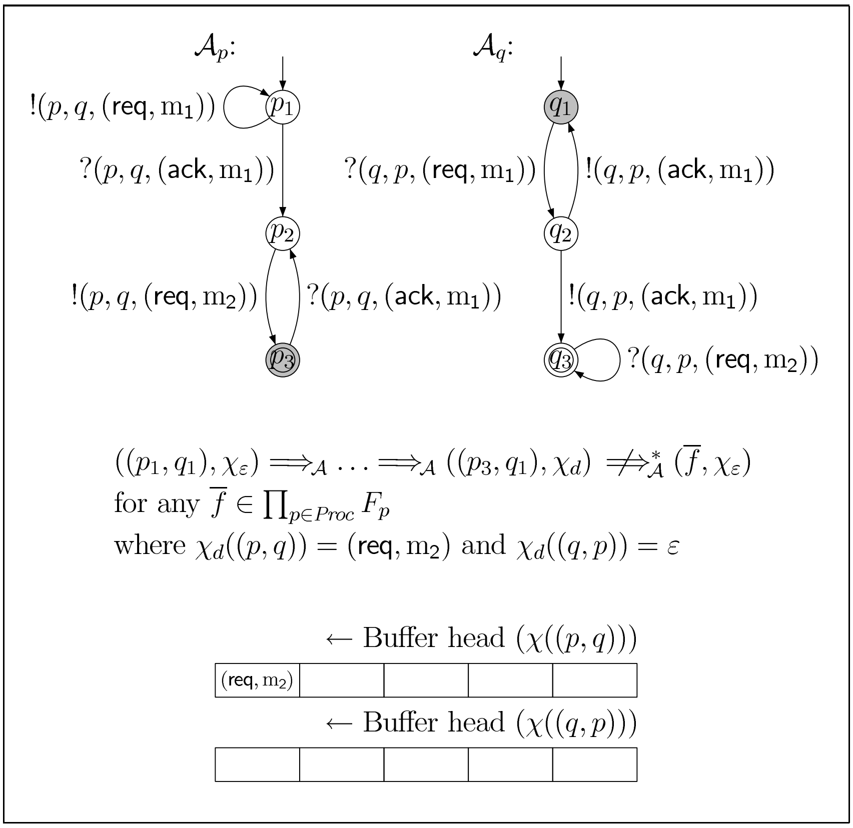

$\mathcal{A}_p$:
$\mathcal{A}_q$:
$!(p,q,(\mathsf{req},\mathrm{m}_1))$
p_1
$?(p,q,(\mathsf{ack},\mathrm{m}_1))$
p_2
$!(p,q,(\mathsf{req},\mathrm{m}_2))$
$?(p,q,(\mathsf{ack},\mathrm{m}_1))$
p_3
q_1
$?(q,p,(\mathsf{req},\mathrm{m}_1))$
$!(q,p,(\mathsf{ack},\mathrm{m}_1))$
q_2
$!(q,p,(\mathsf{ack},\mathrm{m}_1))$
q_3
$?(q,p,(\mathsf{req},\mathrm{m}_2))$
$((p_1,q_1),\chi_\varepsilon) \Longrightarrow_{\mathcal{A}} \ldots \Longrightarrow_{\mathcal{A}} ((p_3,q_1),\chi_d) \not\Longrightarrow^*_{\mathcal{A}} (\overline{f},\chi_\varepsilon)$
for any $\overline{f} \in \prod_{p\in Proc} F_p$
where $\chi_d((p,q)) = (\mathsf{req},\mathrm{m}_2)$ and $\chi_d((q,p)) = \varepsilon$
← Buffer head $(\chi((p,q)))$
$(\mathsf{req},\mathrm{m}_2)$
← Buffer head $(\chi((q,p)))$

Figure 6.4: A deadlock configuration of the CFM from Figure 6.3(e) (page 86)

Definition 6.1.9 (B-bounded word). *Let $B \in \mathbb{N}$. Word $w \in Act^*$ is* B-bounded *if, for any prefix u of w and any $(p,q) \in Ch$, it holds:*

$$0 \;\le\; \sum_{m\in Msg} |u|_{!(p,q,m)} - \sum_{m\in Msg} |u|_{?(q,p,m)} \;\le\; B.$$

This notion is extended to MSCs in the following way. MSC M is called *universally B-bounded* if all words in $Lin(M)$ are B-bounded. MSC M is *existentially B-bounded* if $Lin(M)$ contains at least one B-bounded word.

Consider, e.g., Figure 6.2(d) on page 85. The numbers 1 to 24 in this picture have to be regarded as the sequence in which actions are executed in one (possible) linearization w of the depicted MSC. The numbers in brackets denote the channel contents, where the first component describes number of messages currently stored in channel (p,q) and the second the number of messages in the converse channel. If we take the action sequence corresponding to the numbers 1 to 24, we get a 2-bounded word. This can be verified, by looking at the channel contents: for all channel contents, the number of messages in transit does not exceed the bound $B' = 2$. This implies, that there is $B \ge B'$ such that the MSC $\mathscr{M}(w)$ is universally B-bounded and $B'' \le B'$ such that $\mathscr{M}(w)$ is existentially B''-bounded. In fact, the MSC $\mathscr{M}(w)$ is universally 4-bounded and existentially 2-bounded.

Similar notions are adopted for CFMs, except that for existentially-boundedness, it is required that for every word u of the language, an *equivalent* word $v \approx u$ exists that is B-bounded. The intuition is that bounded channels suffice to accept representatives of the language provided the actions in an CFM are scheduled appropriately, cf. [GKM06, GKM07]. Formally, the notion of bounded CFMs is defined as follows:

Definition 6.1.10 (Bounded CFM ([HMK$^+$05, GKM06])).

a) *CFM $\mathcal{A}$ is* universally b-bounded*, $B \in \mathbb{N}$, if $L(\mathcal{A})$ is a set of B-bounded words. It is* universally bounded *if it is universally B-bounded for some B.*

b) *CFM $\mathcal{A}$ is* existentially B-bounded*, $B \in \mathbb{N}$, if, for every $w \in L(\mathcal{A})$, there is a B-bounded word $w' \in L(\mathcal{A})$ such that $w' \approx w$.*

A further variant of CFMs, as considered in [AEY01, Mor02, Loh03], does not allow for sending control information with a message. Moreover, they only have a single global initial state:

Definition 6.1.11 (Weak CFM). *A CFM $\mathcal{A} = ((\mathcal{A}_p)_{p \in Proc}, \mathcal{I})$ is called* weak *if:*

- $|\mathcal{I}| = 1$ *and*
- *for every two transitions $(s_1, !(p, q, (m_1, \lambda_1)), s_1')$ and $(s_2, !(p, q, (m_2, \lambda_2)), s_2')$, we have $\lambda_1 = \lambda_2$.*

Note that the second item requires that only one message from Λ is used in the CFM. Intuitively, we could say that no synchronization message is used at all, as a weak CFM cannot distinguish between several messages.

Example 6.1.12. Consider the weak CFMs $\mathcal{A}_\mathrm{a}$ and $\mathcal{A}_\mathrm{b}$ depicted in Figure 6.3(a) and (b) (cf. page 86), respectively, which do not use control messages (recall that, formally, there is no distinction between control messages). The CFM $\mathcal{A}_\mathrm{a}$ represents a simple producer-consumer protocol, whereas $\mathcal{A}_\mathrm{b}$ specifies a part of the alternating-bit protocol. Two scenarios that demonstrate a possible behavior of these systems are given by the MSCs M_a and M_b from Figure 6.2(a) and (b), respectively. Indeed, $M_\mathrm{a} \in \mathcal{L}(\mathcal{A}_\mathrm{a})$ and $M_\mathrm{b} \in \mathcal{L}(\mathcal{A}_\mathrm{b})$ (thus, $Lin(M_\mathrm{a}) \subseteq L(\mathcal{A}_\mathrm{a})$ and $Lin(M_\mathrm{b}) \subseteq L(\mathcal{A}_\mathrm{b})$). Observe that $\mathcal{A}_\mathrm{a}$ is deterministic, existentially 1-bounded, and deadlock-free. It is not universally bounded as process p can potentially send arbitrarily many messages to process q before any of these messages is received. In contrast, $\mathcal{A}_\mathrm{b}$ is universally bounded (witnessed by the bound $B = 3$) and also existentially 1-bounded. As stated before, it is also deterministic and deadlock-free.

The CFM $\mathcal{A}_\mathrm{c}$ (cf. Figure 6.3(c)), which is existentially 1-bounded, deadlock-free, and not deterministic, describes the system that is depicted informally in Figure 6.3(d) in terms of a *high-level* MSC: MSCs from $\mathcal{L}(\mathcal{A}_\mathrm{c})$ start with sending a request message from p to q, followed by an arbitrary sequence of further requests, which are sent from p to q, and acknowledgments, sent from q to p (cf. Figure 6.2(c)). Note that $\mathcal{A}_\mathrm{c}$ employs control messages to avoid deadlocks. The idea is that L and R inform the communication partner about which of the nodes at the bottom (left or right) is envisaged next.

The last CFM $\mathcal{A}_\mathrm{d}$ describes a protocol in which process p sends $2n$ ($n > 0$) messages to q and q sends n messages to p. After an initial phase of sending n req messages from process p, process q sends back an ack message after each receive of a req message. Meanwhile, process p waits for the first ack message to arrive. As on process q, all these ack messages are directly answered by a second req, yielding a total number of $2n$ messages sent from process p and n messages sent from process q. To distinguish between these two phases of sending n messages, waiting, and sending another n messages, the CFM makes use of the control messages m_1 and m_2. CFM $\mathcal{A}_\mathrm{d}$ (cf. Figure 6.3(e)) is existentially $\lceil \frac{n}{2} \rceil$-bounded but not universally bounded. It is, moreover, not deterministic and, as we already saw before, may easily deadlock. ◇

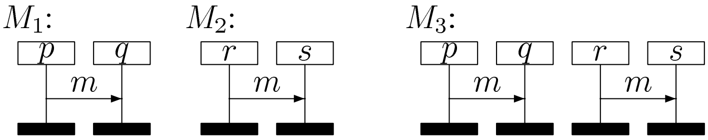


Figure 6.5: Some MSCs

For weak CFMs $\mathcal{A}$, we can identify another closure property. Consider Figure 6.5. If $L(\mathcal{A})$ subsumes the linearizations of the MSCs M_1 and M_2, then those of M_3 will be contained in $L(\mathcal{A})$ as well, as the bilateral interaction between the processes is completely independent. Formally, we define the inference relation $\models \subseteq 2^{Act^*} \times Act^*$ as follows: given a set L of well-formed words and a well-formed word w, $L \models w$ if, for every $p \in Proc$, there is $u \in L$ such that $u \upharpoonright p = w \upharpoonright p$. Here, $w \upharpoonright p \in Act_p^*$ denotes the projection of w onto actions of process p. Indeed, $L(\mathcal{A})$ is closed under $\models$, i.e., $L(\mathcal{A}) \models w$ implies $w \in L(\mathcal{A})$.

A stronger notion, which is satisfied by any weak deadlock-free CFM is as follows. Let $L \subseteq Act^*$ be a set of well-formed words and let u be a proper word (i.e., it is the prefix of some well-formed word). We write $L \models^{df} u$ if, for every $p \in Proc$, there is $w \in L$ such that $u \upharpoonright p$ is a prefix of $w \upharpoonright p$. Language $L \subseteq Act^*$ is closed under $\models^{df}$ if $L \models^{df} w$ implies that w is a prefix of some word in L.

Lemma 6.1.13 ([AEY01, Loh03]). *Let $\mathcal{A}$ be a weak CFM.*

- *$L(\mathcal{A})$ is closed under $\models$.*
- *If $\mathcal{A}$ is deadlock-free, then $L(\mathcal{A})$ is closed under $\models^{df}$.*

Implementability Issues

Next, we collect known results on the relationship between regular languages over Act and CFM languages.

Theorem 6.1.14. *Let $L \subseteq Act^*$ be a set of well-formed words that is closed under $\approx$, and let $B \in \mathbb{N}$. We have the following equivalences:*

a) 1) L is regular.

2) There is a universally bounded CFM $\mathcal{A}$ with $L = L(\mathcal{A})$.

3) There is a deterministic universally bounded CFM $\mathcal{A}$ with $L = L(\mathcal{A})$.

4) There is a universally bounded deadlock-free CFM $\mathcal{A}$ with $L = L(\mathcal{A})$.

b) 1) The set $\{w \in L \mid w \text{ is } B\text{-bounded}\}$ is regular, and for all $w \in L$, there is a B-bounded word w' with $w \approx w'$.

2) There is an existentially B-bounded CFM $\mathcal{A}$ with $L = L(\mathcal{A})$.

c) 1) L is regular and closed under $\models$.

2) There is a universally bounded weak CFM $\mathcal{A}$ with $L = L(\mathcal{A})$.

d) 1) L is regular, closed under $\models$*, and closed under* $\models^{df}$*.*

2) There is a deterministic universally bounded deadlock-free weak CFM $\mathcal{A}$ *with* $L = L(\mathcal{A})$*.*

In all four cases, both directions are effective where L is assumed to be given as a finite automaton.

The equivalences "1) $\Leftrightarrow$ 2) $\Leftrightarrow$ 3)" in Theorem 6.1.14a) go back to [HMK$^+$05], the equivalence "1) $\Leftrightarrow$ 4)" to [BM07]. Theorem 6.1.14b) is due to [GKM06]. Finally, Theorems 6.1.14c) and 6.1.14d) can be attributed to [AEY01, Loh03].

6.2 Learning Communicating Finite-State Machines

As we already saw in Subsection 3.3.1, the algorithm L* synthesizes a minimal DFA from examples given as words. In this section, we intend to adapt this approach such that CFMs are learned from example scenarios that are provided as MSCs. Let us first settle on a user profile, i.e., on some reasonable assumptions about the teacher/oracle that an inference algorithm should respect:

- The user can fix some system characteristics. For example, she might require her system to be deadlock-free, deterministic, or universally bounded.
- She can decide if a given scenario in terms of an MSC is desired or unwanted, thus classify it as positive or negative, respectively.
- She can accept or reject a given system and, in the latter case, come up with an MSC counterexample.

Roughly speaking, the user activity should restrict to classifying and providing MSCs. In contrast, we do not assume that the user can determine if a given system corresponds to the desired system characteristics. Apart from the fact that this would be time too consuming as a manual process, the user often lacks the necessary expertise. Moreover, the whole learning process would get stuck if the user was confronted with a hypothesis that does not match her requirements, but cannot come up with an MSC that is causal for this violation (this is particularly difficult if the system is required to be deadlock-free). So we would like to come up with some *guided* approach that "converges" against a system satisfying the requirements.

The core ingredient of an inference algorithm that matches our user profile shall be the algorithm L*, which synthesizes a minimal DFA from examples given as words. To build a bridge from regular word languages to CFMs, we make use of Theorem 6.1.14, which reveals strong relationships between CFMs and regular word languages over the set *Act* of actions. More specifically, it asserts that one can synthesize:

- a $\underline{\text{deterministic universally bounded CFM}}$ from a regular set of well-formed words that is closed under $\approx$,
- a $\underline{\text{universally bounded deadlock-free CFM}}$ from a regular set of well-formed words that is closed under $\approx$,

- an existentially B-bounded CFM from a regular set of well-formed B-bounded words that is closed under the restriction of $\approx$ to B-bounded words, and
- a universally bounded deadlock-free weak CFM from a regular set of well-formed words that is closed under $\approx$, $\models$, and $\models^{df}$.

Towards learning CFMs, a naïve idea would now be to infer, by means of L*, a regular word language that can be translated into a CFM according to Theorem 6.1.14. The user then has to classify arbitrary words over the alphabet of actions and to deal with hypotheses that have nothing in common with MSC languages. These activities, however, do not match our user profile. Moreover, the user will be confronted with an overwhelming number of membership and equivalence queries that could actually be answered automatically. In fact, words that do not match an execution of a CFM and hypotheses that do not correspond to a CFM could be systematically rejected, without bothering the user. The main principle of our solution will, therefore, be an interface between the user and the program (i.e., the learner) that is based on MSCs only. In other words, the only objects that the user gets to see are MSCs that need to be classified, and CFMs that might already correspond to a desired design model. On the one hand, this facilitates the user activities. On the other hand, we obtain a substantial reduction of membership and equivalence queries. The latter will be underpinned, in Section 9.2, by a practical evaluation (cf. Table 9.1).

Now let us turn to our adapted inference algorithm. Its core will indeed be L*. While L* does not differentiate between words over a given alphabet, however, Theorem 6.1.14 indicates that we need to consider a suitable domain $\mathcal{D} \subseteq Act^*$ containing only well-formed words. Secondly, certain restrictions have to be imposed such that any synthesized CFM recognizes a regular subset of $\mathcal{D}$. For universally-bounded (deadlock-free) CFMs, this might be the class of all well-formed words, whereas for existentially B-bounded CFMs only regular languages of B-bounded words are suitable. In other words, we have to ensure that regular word languages are learned that contain words from $\mathcal{D}$ only. As for any CFM $\mathcal{A}$, $L(\mathcal{A})$ is closed under $\approx$, the regular subsets of $\mathcal{D}$ in addition have to be closed under $\approx$; more precisely, the restriction of $\approx$ to words from $\mathcal{D}$. Similarly, to infer a weak or deadlock-free CFM, we need a regular word language that is closed under $\models$. In our learning setup, this will be captured by a relation $\vdash \subseteq 2^{\mathcal{D}} \times 2^{\mathcal{D}}$ where $L_1 \vdash L_2$ intuitively means that L_1 requires at least one word from L_2. It is not difficult to see that this relation suffices to cover the inference relation $\models$, and as will be shown later, it can be used to capture $\models^{df}$ as well.

Let $\mathfrak{R}_{\mathrm{minDFA}}(\mathcal{D}, \vdash)$ be the class of minimal DFA that recognize a language $L \subseteq \mathcal{D}$ satisfying:

- L is closed under $\approx_{\mathcal{D}} := \approx \cap (\mathcal{D} \times \mathcal{D})$, and
- L is closed under $\vdash$, i.e., $(L_1 \vdash L_2 \wedge L_1 \subseteq L)$ implies $L \cap L_2 \neq \emptyset$.

A learning algorithm tailored to CFMs is now based on the notion of a *learning setup* for a class of CFMs, which provides instantiations of $\mathcal{D}$ and $\vdash$.

Definition 6.2.1 (Learning Setup). *Let $\mathfrak{C}$ be a class of CFMs. A* learning setup *for $\mathfrak{C}$ is a triple $(\mathcal{D}, \vdash, synth)$ where:*

- *$\mathcal{D} \subseteq Act^*$, the* domain*, is a set of well-formed words,*

- $\vdash \subseteq 2^{\mathcal{D}} \times 2^{\mathcal{D}}$ *such that* $L_1 \vdash L_2$ *implies* (L_1 *is finite,* $L_2 \neq \emptyset$*, and* L_2 *is decidable),*
- *synth* : $\mathfrak{R}_{\mathrm{minDFA}}(\mathcal{D}, \vdash) \to \mathfrak{C}$ *is the computable* synthesis function *such that, for each CFM* $\mathcal{A} \in \mathfrak{C}$*, there is* $\mathcal{B} \in \mathfrak{R}_{\mathrm{minDFA}}(\mathcal{D}, \vdash)$ *with* $[L(\mathcal{B})]_{\approx} = L(\mathit{synth}(\mathcal{B})) = L(\mathcal{A})$ *(in particular, synth is injective).*

The final constraint asserts that for any CFM $\mathcal{A}$ in the considered class of CFMs, a minimal DFA $\mathcal{B}$ exists (in the corresponding class of DFA) recognizing the same word language as $\mathcal{A}$ modulo $\approx$.

Given the kind of learning setup that we will consider, we now discuss some necessary changes to the algorithm L*. As L* works within the class of arbitrary DFA over *Act*, conjectures may be proposed whose languages are not subsets of $\mathcal{D}$, or violate the closure properties for $\approx$ and $\vdash$ (or both). To avoid the generation of such incorrect hypothesized automata, the language inclusion problem (is the language of a given DFA included in $\mathcal{D}$?) and the closure properties in question are required to be *constructively decidable*. This means that each of these problems is decidable and that in case of a negative result, a *reason* of its failure, i.e., a counterexample, can be computed. Accordingly, we require that the following properties hold for DFA $\mathcal{B}$ over *Act*:

(D1) The problem whether $L(\mathcal{B}) \subseteq \mathcal{D}$ is decidable and if $L(\mathcal{B}) \not\subseteq \mathcal{D}$, one can compute some $w \in L(\mathcal{B}) \setminus \mathcal{D}$. We then say that INCLUSION($\mathcal{D}$) is *constructively decidable*.

(D2) If $L(\mathcal{B}) \subseteq \mathcal{D}$, it is decidable whether $L(\mathcal{B})$ is closed under $\approx_{\mathcal{D}}$. If not, one can compute $w, w' \in \mathcal{D}$ such that $w \approx_{\mathcal{D}} w'$, $w \in L(\mathcal{B})$, and $w' \notin L(\mathcal{B})$. We then say that the problem EQCLOSURE($\mathcal{D}$) is *constructively decidable*.

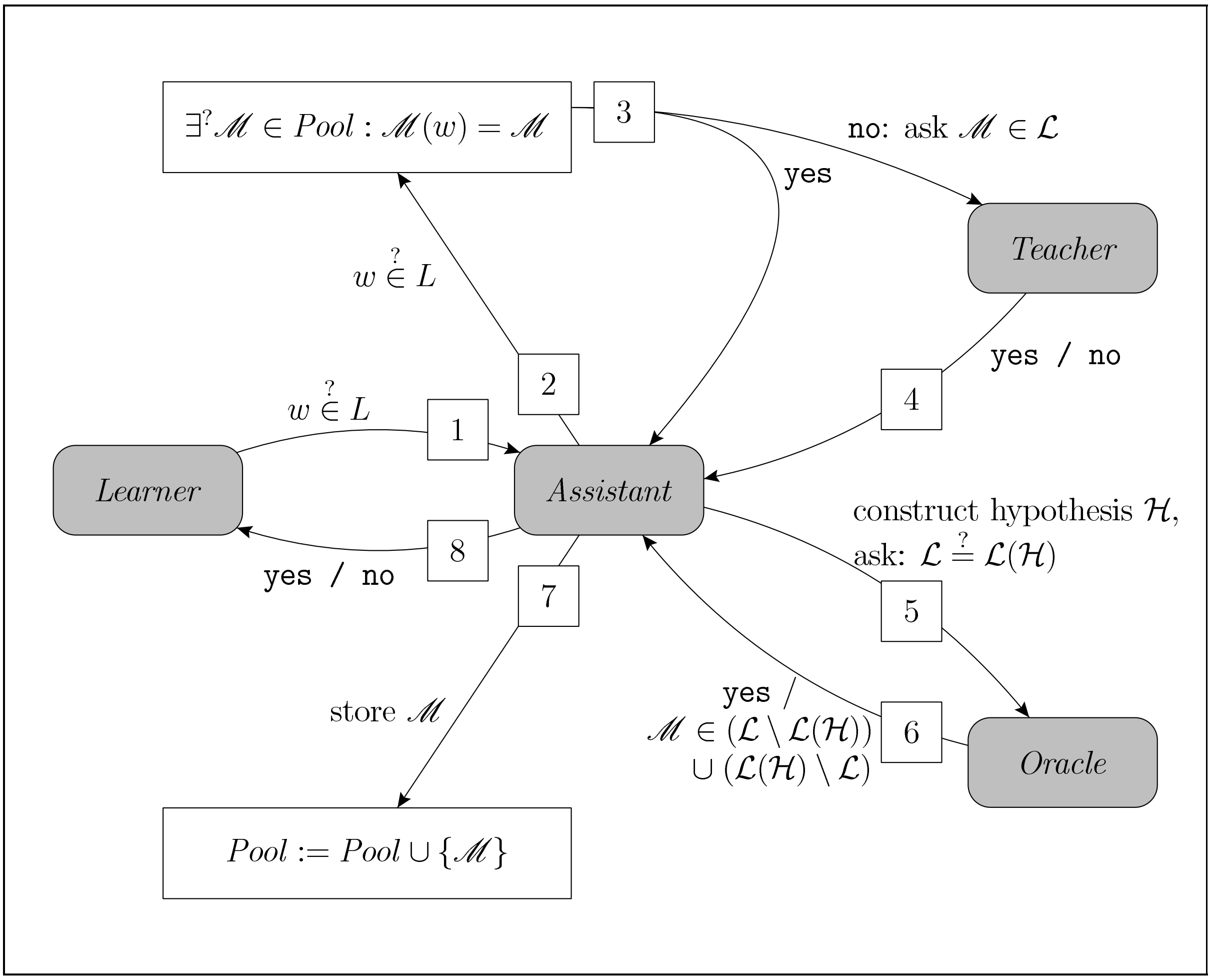
(D3) If $L(\mathcal{B}) \subseteq \mathcal{D}$ is closed under $\approx_{\mathcal{D}}$, it is decidable whether $L(\mathcal{B})$ is closed under $\vdash$. If not, one can compute $(L_1, L_2) \in \vdash$ (hereby, L_2 shall be given in terms of a decision algorithm that checks a word for membership) such that $L_1 \subseteq L(\mathcal{B})$ and $L(\mathcal{B}) \cap L_2 = \emptyset$. We then say that INFCLOSURE($\mathcal{D}, \vdash$) is *constructively decidable*.

Let us now generalize Angluin's algorithm to cope with the extended setting, and let ($\mathcal{D}, \vdash$, *synth*) be a learning setup for some class $\mathfrak{C}$ of CFMs. The main changes in Angluin's algorithm concern the processing of membership queries as well as the treatment of hypotheses. For the following description, we refer to Table 6.1, depicting the pseudocode of EXTENDED-L*, our extension of L*, Table 6.2 which contains a modified table-update function that is invoked by this extension of L*, and Figure 6.6, which schematically describes the new learning algorithm.

The *Teacher* will provide/classify MSCs rather than words. Moreover, the equivalence test will be performed, by the *Oracle*, on the basis of a CFM rather than on the basis of a DFA. The *Oracle* will also provide counterexamples in terms of MSCs (cf. Figure 6.6(5, 6)).

To undertake an equivalence test, knowledge of the target model is required as in every other learning based technique for inferring design models. Simulating and testing are possibilities to converge to a correct system implementation. In the implementation of our approach [BKKL08a] we provide such means to ease the user's burden.

To realize these changes, we exploit a new table-update function EXTENDED-T-UPDATE (cf. Table 6.2). Therein, membership queries are filtered: a query $w \notin \mathcal{D}$ is considered immediately as negative, without presenting it to the *Teacher* (lines 2, 3 of Table 6.2 and Figure 6.6(1, 8)). Note that in Figure 6.6 a new component (called the *Assistant*) takes over the part of the filter. It also prevents equivalent membership queries from being



$\exists^? \mathscr{M} \in Pool : \mathscr{M}(w) = \mathscr{M}$
3
no: ask $\mathscr{M} \in \mathcal{L}$
yes
Teacher
$w \overset{?}{\in} L$
2
yes / no
4
$w \overset{?}{\in} L$
1
Learner
Assistant
construct hypothesis $\mathcal{H}$,
ask: $\mathcal{L} \overset{?}{=} \mathcal{L}(\mathcal{H})$
8
yes / no
7
5
store $\mathscr{M}$
yes /
$\mathscr{M} \in (\mathcal{L} \setminus \mathcal{L}(\mathcal{H}))$
$\cup (\mathcal{L}(\mathcal{H}) \setminus \mathcal{L})$
6
Oracle
$Pool := Pool \cup \{\mathscr{M}\}$

Figure 6.6: Components of EXTENDED-L* and their interactions

displayed several times. Faced with a query $w \in \mathcal{D}$, the MSC $\mathscr{M}(w)$ is displayed to the *Teacher* (we call this a *user query*, cf. also Figure 6.6(2, 3)) as long as there has not been an equivalent word $w' \approx w$ that already has been classified before. His verdict will then determine the table entry for w (line 9 of Table 6.2 and Figure 6.6(4)). Once a user query has been processed for a word $w \in \mathcal{D}$, queries $w' \in [w]_{\approx_\mathcal{D}}$ must be answered equivalently. They are thus not forwarded to the *Teacher* (lines 6, 7 of Table 6.2 and Figure 6.6(2, 3)) anymore. Therefore, MSCs that have already been classified are memorized in a set *Pool* (line 10 of Table 6.2 and Figure 6.6(7)).

Once table $\mathcal{T}$ is closed and consistent, a hypothesized DFA $\mathcal{H}_\mathcal{T}$ is determined as usual. We then proceed as follows (cf. Table 6.1):

(i) If $L(\mathcal{H}_\mathcal{T}) \not\subseteq \mathcal{D}$, compute a word $w \in L(\mathcal{H}_\mathcal{T}) \setminus \mathcal{D}$ and modify the table $\mathcal{T}$ accordingly by invoking EXTENDED-T-UPDATE (lines 19–23).

(ii) If $L(\mathcal{H}_\mathcal{T}) \subseteq \mathcal{D}$ but $L(\mathcal{H}_\mathcal{T})$ is not closed under $\approx_\mathcal{D}$, then compute $w, w' \in \mathcal{D}$ such that $w \approx_\mathcal{D} w'$, $w \in L(\mathcal{H}_\mathcal{T})$, and $w' \notin L(\mathcal{H}_\mathcal{T})$; perform the membership queries for $[w]_\approx$. As these queries are performed in terms of an MSC by displaying $\mathscr{M}(w)$ to the *Teacher*, it is guaranteed that they are answered uniformly (lines 25–29).

(iii) If $L(\mathcal{H}_\mathcal{T})$ is the union of $\approx_\mathcal{D}$-equivalence classes but not closed under $\vdash$, then compute $(L_1, L_2) \in \vdash$ such that $L_1 \subseteq L(\mathcal{H}_\mathcal{T})$ and $L(\mathcal{H}_\mathcal{T}) \cap L_2 = \emptyset$; perform user queries for every word from L_1 (displaying the corresponding MSCs to the *Teacher*); if all these queries are answered positively, the *Teacher* is asked to specify an MSC that comes with a linearization w from L_2. The word w will be declared "positive". Recall that L_2 is a decidable language (and we assume that the decision algorithm is available) so that all MSCs $\mathscr{M}$ with $\mathit{Lin}(\mathscr{M}) \cap L_2 \neq \emptyset$ can be enumerated until a suitable MSC is selected (lines 31–42).

If, for a hypothesized DFA $\mathcal{H}_\mathcal{T}$, we have $L(\mathcal{H}_\mathcal{T}) \subseteq \mathcal{D}$ and $L(\mathcal{H}_\mathcal{T})$ is closed under both $\approx_\mathcal{D}$ and $\vdash$, then an equivalence query is performed on $\mathit{synth}(\mathcal{H}_\mathcal{T})$, the CFM that is synthesized from the hypothesized DFA. In case a counterexample MSC $\mathscr{M}$ is provided, the table has to be complemented accordingly by a linearization of $\mathscr{M}$ (lines 44–51). Otherwise, $\mathit{synth}(\mathcal{H}_\mathcal{T})$ is returned as the desired CFM (lines 52, 53).

Theorem 6.2.2. *Let $\mathfrak{C}$ be a class of CFMs, let $(\mathcal{D}, \vdash, \mathit{synth})$ be a learning setup for $\mathfrak{C}$, and let $\mathcal{A} \in \mathfrak{C}$. If the Teacher classifies/provides MSCs in conformance with $\mathcal{L}(\mathcal{A})$, then invoking* EXTENDED-L$^*(\mathcal{D}, \vdash, \mathit{synth})$ *eventually returns a CFM $\mathcal{A}' \in \mathfrak{C}$ such that $L(\mathcal{A}') = L(\mathcal{A})$.*

Proof: We fix a class $\mathfrak{C}$ of CFMs and a learning setup $(\mathcal{D}, \vdash, \mathit{synth})$ for $\mathfrak{C}$. Moreover, let $\mathcal{A} \in \mathfrak{C}$. By the definition of a learning setup, there exists a DFA $\mathcal{B} \in \mathfrak{R}_{\mathrm{minDFA}}(\mathcal{D}, \vdash)$ with $[L(\mathcal{B})]_\approx = L(\mathit{synth}(\mathcal{B})) = L(\mathcal{A})$. We suppose that the *Teacher* classifies/provides MSCs in accordance with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathit{synth}(\mathcal{B}))$. On invoking EXTENDED-L$^*(\mathcal{D}, \vdash, \mathit{synth})$, a word $w \in (U \cup U\mathit{Act})V$ is classified by the table function T depending on whether $w \in \mathcal{D}$ and $\mathscr{M}(w) \in \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathit{synth}(\mathcal{B}))$. More precisely, $T(w) = +$ iff $w \in L(\mathcal{B})$, i.e., we actually perform L*, and the *Teacher* acts in conformance with $L(\mathcal{B})$. The differences to the basic version of Angluin's algorithm are that (i) not every hypothesis $\mathcal{H}_{(T,U,V)}$ is forwarded to the *Teacher* (in that case, counterexamples can be generated automatically), and (ii) we may add, in lines 28 and 34, several words (and their prefixes) to the table at one go. This is, however, a modification that preserves the validity of Theorem 3.3.3 from page 37. Consequently, when the equivalence test succeeds (line 52), then the algorithm outputs a CFM $\mathcal{A}' = \mathit{synth}(\mathcal{H}_{(T,U,V)})$ with $L(\mathcal{A}') = L(\mathcal{A})$. □

Having introduced the notion of a learning setup and proved the correctness of our extended learning algorithm for CFMs, it is sensible to define the *learnability* of CFM classes.

Definition 6.2.3 (Learnability of classes of CFMs). *A class $\mathfrak{C}$ of CFMs is* learnable *if there is a learning setup for $\mathfrak{C}$.*

The sequel of this section is devoted to identify learnable classes of CFMs. To this purpose, we have to determine a learning setup for each such class.

A note on the complexity. The total running time of the extended algorithm can only be considered wrt. a concrete learning setup. In particular, it heavily depends on the complexity of the synthesis of a CFM from a given minimal DFA, which tends to be very high. When studying this issue below for several learning setups, we will therefore assume that an equivalence check is performed on the basis of the minimal DFA itself rather than on a synthesized CFM (cf. line 44 in Table 6.1). This lowers the running time of the algorithm considerably and, at the same time, is a reasonable assumption, as in all learning setups we provide below, the minimal DFA faithfully simulates all executions of the synthesized CFM (up to a channel bound when considering the case of existential bounds). So let us in the following assume the synthesis function to need constant time.

```
EXTENDED-L*($\mathcal{D}, \vdash, synth$):
U := {ε}; V := {ε}; T is defined nowhere;
Pool := ∅;
EXTENDED-T-UPDATE();
repeat
        while $(T, U, V)$ is not (closed and consistent)
            do
                if $(T, U, V)$ is not consistent then
                  find $u, u' \in U, a \in Act$, and $v \in V$ such that $row(u) = row(u')$ and
                                                              $row(ua)(v) \neq row(u'a)(v)$;
                  $V := V \cup \{av\}$;
                  EXTENDED-T-UPDATE();
                if $(T, U, V)$ is not closed then
                  find $u \in U$ and $a \in Act$ such that $row(ua) \neq row(u')$ for all $u' \in U$;
                  $U := U \cup \{ua\}$;
                  EXTENDED-T-UPDATE();
        /* $(T, U, V)$ is both closed and consistent */
        $\mathcal{H} := \mathcal{H}_{(T,U,V)}$;
        /* check closedness properties for $\approx_{\mathcal{D}}$ and $\vdash$ */
        if $L(\mathcal{H}) \not\subseteq \mathcal{D}$
          then
               compute $w \in L(\mathcal{H}) \setminus \mathcal{D}$;
               $U := U \cup pref(w)$;
               EXTENDED-T-UPDATE();
          else
               if $L(\mathcal{H})$ is not $\approx_{\mathcal{D}}$-closed
                 then
                      compute $w, w' \in \mathcal{D}$ such that $w \approx_{\mathcal{D}} w'$, $w \in L(\mathcal{H})$, and $w' \notin L(\mathcal{H})$;
                      $U := U \cup pref(w) \cup pref(w')$;
                      EXTENDED-T-UPDATE();
                 else
                      if $L(\mathcal{H})$ is not $\vdash$-closed
                        then
                             compute $(L_1, L_2) \in \vdash$ such that $L_1 \subseteq L(\mathcal{H})$ and $L(\mathcal{H}) \cap L_2 = \emptyset$;
                             $U := U \cup pref(L_1)$;
                             EXTENDED-T-UPDATE();
                             if $T(w) = +$ for all $w \in L_1$ then
                               $\mathscr{M} := getMSCFromTeacher(L_2)$;
                               choose $w \in Lin(\mathscr{M}) \cap L_2$;
                               $U := U \cup pref(w)$;
                               $T(w) := +$;
                               $Pool := Pool \cup \{\mathscr{M}\}$;
                               EXTENDED-T-UPDATE();
                        else
                             do equivalence test for $synth(\mathcal{H}_{(T,U,V)})$;
                             if equivalence test fails then
                               counterexample $\mathscr{M}$ is provided, classified as $parity \in \{+, -\}$;
                               choose $w \in Lin(\mathscr{M}) \cap \mathcal{D}$;
                               $U := U \cup pref(w)$;
                               $T(w) := parity$;
                               $Pool := Pool \cup \{\mathscr{M}\}$;
                               EXTENDED-T-UPDATE();
until equivalence test succeeds;
return $synth(\mathcal{H})$;
```

Table 6.1: EXTENDED-L*: The extension of Angluin's algorithm for learning CFMs

```
EXTENDED-T-UPDATE():
for $w \in (U \cup UAct)V$ such that $T(w)$ is not defined
    if $w \notin \mathcal{D}$
      then $T(w) := -$;
      else if $\mathscr{M}(w) \in Pool$
              then
                     choose $w' \in [w]_{\approx_\mathcal{D}}$ such that $T(w')$ is defined;
                     $T(w) := T(w')$;
              else
                     $T(w) := getClassificationFromTeacher(\mathscr{M}(w))$;
                     $Pool := Pool \cup \{\mathscr{M}(w)\}$;
```

Table 6.2: Function for updating the table in EXTENDED-L*

We can now state our first learnability result:

Theorem 6.2.4. *Universally bounded CFMs are learnable.*

Proof: Let $\mathfrak{C}$ denote the class of deterministic universally bounded CFMs. To show that $\mathfrak{C}$ is learnable, we need to determine a learning setup $(\mathcal{D}, \vdash, synth)$ for $\mathfrak{C}$. First observe that $\models$ needs not be instantiated for this class (cf. Theorem 6.1.14a)). Let $\mathcal{D}$ be the set of well-formed words over Act. By Theorem 6.1.14a), there is a computable mapping $synth$ that transforms any regular set L of well-formed words that is closed under $\approx_\mathcal{D} = \approx$ (say, given in terms of a finite automaton) into a CFM $\mathcal{A}$ such that $L(\mathcal{A}) = L$. To show that $(\mathcal{D}, \emptyset, synth)$ is indeed a learning setup, it remains to establish that the problems INCLUSION($\mathcal{D}$), EQCLOSURE($\mathcal{D}$), and INFCLOSURE($\mathcal{D}, \emptyset$) are constructively decidable. Decidability of INCLUSION($\mathcal{D}$) and EQCLOSURE($\mathcal{D}$) has been shown in [HMK+05]. For DFA, these problems are actually solvable in linear time. The decidability of INFCLOSURE($\mathcal{D}, \emptyset$) is trivial. □

Together with Theorem 6.1.14a), we immediately obtain the learnability of two subclasses:

Theorem 6.2.5. *Deterministic universally bounded CFMs and, moreover, universally bounded deadlock-free CFMs are learnable.*

Now let us have a closer look at the complexity of our algorithm, when it is instantiated with the learning setup that we developed in the proof of Theorem 6.2.4. In the best case, we start with a deterministic CFM. In the following, let m denote the maximal number of events of an MSC that is either provided or to be classified by the user (*Teacher* or *Oracle*).

Theorem 6.2.6. *Let $\mathfrak{C}$ be the class of deterministic universally bounded CFMs and let $\mathcal{A} \in \mathfrak{C}$ be universally B-bounded. The number of equivalence queries needed to infer a CFM $\mathcal{A}' \in \mathfrak{C}$ with $L(\mathcal{A}) = L(\mathcal{A}')$ is at most $(|\mathcal{A}| \cdot |Msg| + 1)^{B \cdot |Proc|^2 + |Proc|}$. Moreover, the number of membership queries and the overall running time is polynomial in $|\mathcal{A}|$, $|Msg|$, and m, and it is exponential in B and $|Proc|$.*

Proof: Suppose $\mathcal{A} \in \mathfrak{C}$ is the input CFM. Without loss of generality, we assume that the synchronization messages from Λ that are used in $\mathcal{A}$ are precisely the local states of $\mathcal{A}$. Then, the number of states of the unique minimal DFA $\mathcal{B}$ satisfying $L(synth(\mathcal{B})) = L(\mathcal{A})$

is bounded by $C = |\mathcal{A}|^{|Proc|} \cdot (|Msg| \cdot |\mathcal{A}| + 1)^{B\cdot|Proc|^2}$. The first factor is the number of global states of $\mathcal{A}$, whereas the second factor contributes the number of possible channel contents ($|Msg| \cdot |\mathcal{A}|$ being the number of messages). Hence, C constitutes an upper bound for the number of equivalence queries. We will now calculate the number of membership queries, which is bounded by the size of the table that we obtain when the algorithm terminates. Note first that the size of *Act* is bounded by $2|Proc|^2 \cdot |Msg|$. During a run of the algorithm, the size of V is bounded by C, as the execution of program line 10 always comes with creating a new state. The set U can increase at most C-times, too. The number of words that are added to U in line 22 can be bounded by $2C$. The length of words w and w', as added in line 28, can likewise be bounded by $2C$. The number of words added in line 48 depends on the size of a counterexample that is provided by the *Oracle*. Note that lines 34 and 39 are of no importance here because, as mentioned before, $\vdash$ was not instantiated for this learning setup. Summarizing, the number of membership queries is in $\mathcal{O}((C^3 + mC^2) \cdot |Act|)$ with m the maximal number of events of an MSC that is provided/classified by the user (*Teacher* or *Oracle*). As for a given minimal DFA $\mathcal{H}$, one can detect in polynomial time if $L(\mathcal{H}) \subseteq \mathcal{D}$ and if $L(\mathcal{H})$ is $\approx_{\mathcal{D}}$-closed, the overall running time of the algorithm is polynomial in $|Msg|$, m, and $|\mathcal{A}|$, and it is exponential in $|Proc|$ and B. □

The following theorem states that the complexity is higher when we act on the assumption that the CFM to learn is non-deterministic.

Theorem 6.2.7. *Let $\mathfrak{C}$ be the class of universally bounded (deadlock-free) CFMs and let $\mathcal{A} \in \mathfrak{C}$ be universally B-bounded. The number of equivalence queries needed to infer a CFM $\mathcal{A}' \in \mathfrak{C}$ with $L(\mathcal{A}) = L(\mathcal{A}')$ is at most $2^{(|\mathcal{A}|\cdot|Msg|+1)^{B\cdot|Proc|^2+|Proc|}}$. Moreover, the number of membership queries and the overall running time is polynomial in m, exponential in $|\mathcal{A}|$ and $|Msg|$, and doubly exponential in B and $|Proc|$.*

Proof: We follow the proof of Theorem 6.2.6. As $\mathcal{A}$ can be non-deterministic, however, we have to start from the assumption that the number of states of the unique minimal DFA $\mathcal{B}$ satisfying $L(synth(\mathcal{B})) = L(\mathcal{A})$ is bounded by $2^{(|\mathcal{A}|\cdot|Msg|+1)^{B\cdot|Proc|^2+|Proc|}}$. □

Theorem 6.2.8. *For $B \in \mathbb{N}$, existentially B-bounded CFMs are learnable.*

Let $\mathfrak{C}$ be the class of existentially B-bounded CFMs. We can provide a learning setup such that, for all $\mathcal{A} \in \mathfrak{C}$, the number of equivalence queries needed to infer an existentially B-bounded CFM $\mathcal{A}' \in \mathfrak{C}$ with $L(\mathcal{A}) = L(\mathcal{A}')$ is at most $2^{(|\mathcal{A}|\cdot|Msg|+1)^{B\cdot|Proc|^2+|Proc|}}$. Moreover, the number of membership queries and the overall running time are polynomial in m, exponential in $|Msg|$ and $|\mathcal{A}|$, and doubly exponential in B and $|Proc|$.

Proof: To obtain a learning setup $(\mathcal{D}, \vdash, synth)$ for $\mathfrak{C}$, let $\mathcal{D}$ be the set of B-bounded well-formed words over *Act*. As in the previous proof, $\vdash$ is not needed, i.e., we set $\vdash$ to be $\emptyset$. By Theorem 6.1.14c), there is a computable mapping *synth* that transforms any regular set L of B-bounded well-formed words that is closed under $\approx_{\mathcal{D}}$ into a CFM $\mathcal{A}$ with $L(\mathcal{A}) = [L]_{\approx}$. In order to show that $(\mathcal{D}, \emptyset, synth)$ is a learning setup it remains to show that the problems Inclusion($\mathcal{D}$) and EqClosure($\mathcal{D}$) are constructively decidable. This is shown by a slight modification of the algorithm in [HMK$^+$05] for universally bounded languages. This goes as follows:

Let $\mathcal{B} = (Q, \{q_0\}, \delta, F)$ be a minimal DFA over *Act*. A state $s \in Q$ is called *productive* if there is a path from s to some final state. We successively label any productive state with a channel content, i.e., a function $\chi_s : Ch \to Msg^*$ will be associated to any state $s \in Q$ such that:

(i) The initial state q_0 and any final state $q \in F$ are equipped with χ_ε, mapping any channel to the empty word.

(ii) If $s, s' \in Q$ are productive states and $\delta(s, !(p, q, m)) = s'$, then $\chi_{s'} = \chi_s[(p, q) := m \cdot \chi_s((p, q))]$, i.e., m is appended to channel (p, q).

(iii) If $s, s' \in Q$ are productive states and $\delta(s, ?(q, p, m)) = s'$, then $\chi_s = \chi_{s'}[(p, q) := \chi_{s'}((p, q)) \cdot m]$, i.e., m is removed from the channel (p, q).

$L(\mathcal{B})$ is a set of well-formed words iff there exists a labeling of productive states with channel functions satisfying (i)–(iii). If a state-labeling violates one of the conditions (i)–(iii), then this is due to a word that is not well-formed. This word acts as a counterexample for the Inclusion($\mathcal{D}$) problem. For example, a clash in terms of productive states $s, s' \in Q$ such that $\delta(s, !(p, q, m)) = s'$ and $\chi_{s'}((p, q)) \neq m \cdot \chi_s((p, q))$ gives rise to a path from the initial state to a final state via the transition $(s, !(p, q, m), s')$ that is labeled with a non-well-formed word. This word then acts as a counterexample. Thus, Inclusion($\mathcal{D}$) is constructively decidable.

To show decidability of EqClosure($\mathcal{D}$), consider a further (decidable) property:

(iv) Suppose $\delta(s, a) = s_1$ and $\delta(s_1, b) = s_2$ with $a \in Act_p$ and $b \in Act_q$ for some $p, q \in Proc$ satisfying $p \neq q$. If not $(|\chi_s((q, q'))| = B$ and $b = !(q, q', m)$ for some $q' \in Proc$ and $m \in Msg)$ and, moreover, $(a = !(p, q, m)$ and $b = ?(q, p, m)$ for some $m \in Msg)$ implies $0 < |\chi_s((p, q))|$, then there exists a state $s_1' \in Q$ such that $\delta(s, b) = s_1'$ and $\delta(s_1', a) = s_2$.

This *diamond* property describes in which case two successive actions a and b may be permuted. It follows that the set $L(\mathcal{B})$ of well-formed words is closed under $\approx_{\mathcal{D}}$ iff condition (iv) holds. This is thanks to the fact that we deal with a deterministic automaton. In case item (iv) is violated, let w and w' be words of the form $uabv$ and $ubav$, respectively. These words prove that $L(\mathcal{B})$ is not closed under $\approx_{\mathcal{D}}$. Thus, EqClosure($\mathcal{D}$) is constructively decidable. Note that both Inclusion($\mathcal{D}$) and EqClosure($\mathcal{D}$) are actually solvable in linear time.

To establish the bounds on the overall running time and on the number of equivalence and membership queries, we refer to the considerations in the proofs of Theorems 6.2.4 and 6.2.5. □

Theorem 6.2.9. *Deterministic universally bounded deadlock-free weak CFMs are learnable.*

Let $\mathfrak{C}$ be the class of deterministic universally bounded deadlock-free weak CFMs. We can provide a learning setup such that, for all universally B-bounded CFMs $\mathcal{A} \in \mathfrak{C}$, the number of equivalence queries needed to infer an equivalent CFM $\mathcal{A}' \in \mathfrak{C}$ is at most $(|\mathcal{A}| \cdot |Msg| + 1)^{B \cdot |Proc|^2}$. Moreover, the number of membership queries and the overall running time are polynomial in $|Msg|$ and m, exponential in $|\mathcal{A}|$, and doubly exponential in B and $|Proc|$.

Proof: Let $\mathcal{D}$ be the set of all well-formed words. Unlike the previous proofs, we need an inference relation $\vdash \neq \emptyset$ that respects both $\models$ and $\models^{df}$. Let $\vdash$ be the union of:

$$\{(L, \{w\}) \mid L \models w \text{ and } L \subseteq \mathcal{D} \text{ is finite}\}$$

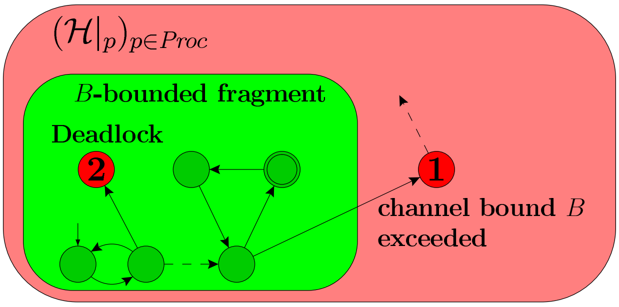


Figure 6.7: Schematic view of error cases for proof of Theorem 6.2.9

(which reflects $\models$) and:

$$\{(L_1, L_2) \mid L_1 \subseteq \mathcal{D} \text{ is finite and} \\ L_2 = \{uv \in \mathcal{D} \mid L_1 \models^{df} u\} \neq \emptyset\}$$

(which reflects $\models^{df}$). Theorem 6.1.14e) provides the required synthesis function.

Decidability of InfClosure($\mathcal{D}, \vdash$) has been shown in [AEY01, Theorem 3]. Alur et al. provide an EXPSPACE-algorithm for bounded high-level MSCs, which reduces the problem at hand to a decision problem for finite automata with an $\approx$-closed language. The latter is actually in PSPACE. The first step is to construct from the given $\approx$-closed DFA $\mathcal{H}$ a (component-wise) minimal and deterministic weak CFM $\mathcal{A}'$, by simply taking the projections $\mathcal{H}\!\upharpoonright_p$ of $\mathcal{H}$ onto Act_p for any $p \in Proc$, determinizing and minimizing them. Then, $L(\mathcal{H})$ is closed under both $\models$ and $\models^{df}$ iff $\mathcal{A}'$ is a deadlock-free CFM such that $L(\mathcal{A}') = L(\mathcal{H})$. From $\mathcal{H}$, we can, moreover, compute a bound B such that any run of $\mathcal{A}'$ exceeding the buffer size B cannot correspond to a prefix of some word in $L(\mathcal{H})$.

Thus, a partial run of $\mathcal{A}'$ that either:

- exceeds the buffer size B (i.e., it is not B-bounded; cf. Fig. 6.7, node 1), or
- respects the buffer size B, but results in a deadlock configuration (cf. Fig. 6.7, node 2),

gives rise to a proper word $u \in Act^*$ that is implied by $\mathcal{H}$ wrt. $\models^{df}$, i.e., $L(\mathcal{H})$ must actually contain a well-formed completion uv of u. Obviously, one can decide if a word is such a completion of u. The completions of u form one possible L_2. It remains to specify a corresponding set L_1 for u. By means of $\mathcal{H}$, we can, for any $p \in Proc$, compute a word $w_p \in L(\mathcal{H})$ such that $u \upharpoonright p$ is a prefix of $w_p \upharpoonright p$. We set $L_1 = \{w_p \mid p \in Proc\}$.

Finally, suppose that, in $\mathcal{A}'$, we could neither find a prefix exceeding the buffer size B nor a reachable deadlock configuration in the B-bounded fragment. Then, we still have to check if $\mathcal{A}'$ recognizes $L(\mathcal{H})$. If this is not the case, one can compute a (B-bounded) word $w \in L(\mathcal{A}') \setminus L(\mathcal{H})$ such that $L(\mathcal{A}') \models w$. Setting $L_2 = \{w\}$, a corresponding set L_1 can be specified as $\{w_p \mid p \in Proc\}$, as above.

Let us turn to the complexity of this particular learning setup. We can partly follow the proof of Theorem 6.2.4. As lines 31–42 come into play, however, the complexity estimation is more complicated. The number of equivalence queries is bounded by $C = (|\mathcal{A}| \cdot |Msg| + 1)^{B\cdot|Proc|^2+|Proc|}$ where $\mathcal{A}$ is the CFM at hand. To compute the number of membership queries, we have to take into account the number of words that are added to U in lines 34 and 39 in the pseudocode of the algorithm. To this aim, note that the number of global states of the deterministic weak CFM $\mathcal{A}'$ that we compute above is bounded by $2^{C \cdot |\mathit{Proc}|}$. Moreover, the number of possible channel contents is bounded by $(|\mathit{Msg}| + 1)^{B' \cdot |\mathit{Proc}|^2}$ where $B' = C$ is the maximal number of states of $\mathcal{H}$. Hence,

$$N := 2^{C \cdot |\mathit{Proc}|} \cdot (|\mathit{Msg}| + 1)^{C \cdot |\mathit{Proc}|^2}$$

is an upper bound for the number of configurations of $\mathcal{A}'$ that we have to consider. Moreover, N constitutes a bound on the length of words from L_1 as far as it concerns $\models^{\mathit{df}}$. In turn, L_1 contains $|\mathit{Proc}|$ many words. Now let us turn towards $\models$. To obtain a word w from $L(\mathcal{A}') \setminus L(\mathcal{H})$, we build the product of the complement automaton of $\mathcal{H}$, which is of the same size as $\mathcal{H}$, and the configuration automaton of $\mathcal{A}'$. Thus, the length of w, which constitutes one possible L_2, can be bounded by $C \cdot N$ so that $\mathit{pref}(L_1)$ contains at most $|\mathit{Proc}| \cdot C \cdot N$ words. Therefore, the number of membership queries is in $\mathcal{O}((|\mathit{Proc}| \cdot N \cdot C^3 + mC^2) \cdot |\mathit{Act}|)$. Furthermore, we deduce that the overall running time of the algorithm is polynomial in $|\mathit{Msg}|$ and m, exponential in $|\mathcal{A}|$, and doubly exponential in B and $|\mathit{Proc}|$. □

Theorem 6.1.14d) provides a characterization of (deterministic) universally bounded weak CFMs in terms of regular word languages. Let $\mathcal{D}$ be the set of all well-formed words and let $\vdash$ be given by $\{(L, \{w\}) \mid L \models w \text{ and } L \subseteq \mathcal{D} \text{ is finite}\}$ reflecting $\models$. Unfortunately, the problem InfClosure($\mathcal{D}, \vdash$) is undecidable (cf. [AEY01]) so that the above approach does not work for this particular class of CFMs. One might argue that universally bounded weak CFMs are still learnable, as their regular word languages can be inferred with L*. But an approach that relies solely on L* requires additional expertise from a user. The latter has to make sure by herself that the final hypothesis corresponds to a universally bounded weak CFM. But if we assume that the user needs some guidance and, at the beginning, has an incomplete idea of her system, then we have, for the moment, no means to infer universally bounded weak CFMs.

Note that the complexity of our algorithms is, in most cases, not worse than that of L* if we refer to the size of the underlying minimal DFA. The only instance where the complexity is exponentially higher compared to L* (wrt. the size of the minimal DFA) is reported in Theorem 6.2.9. This explosion, however, accounts for the automatic test of hypotheses for deadlocks, which, otherwise, would have to be carried out manually by the user.

6.3 Partial-Order Learning

We are now going to introduce *partial-order learning* as an application of the learning of congruence-closed languages approach presented in Chapter 5. In our current setting, the normal forms *pnf* and *snf* are defined over an action alphabet *Act*, a domain $\mathcal{D} \subseteq \mathit{Act}^*$ which only contains proper, well-formed words over *Act*, and the equivalence relation $\approx_{\mathcal{D}}$. Note that thereby, the existence of elements which are not in $\mathit{pref}(\mathcal{D})$ or $\mathit{suff}(\mathcal{D})$ is guaranteed (so the additional auxiliary symbol $\perp$ from Chapter 5 is not needed here).

Let, moreover, $(\mathcal{D}, \approx, \vdash)$ be a learning setup for class $\mathfrak{C}$ of CFMs and let $\mathit{pnf}, \mathit{snf} : \mathit{Act}^* \to \mathit{Act}^*$. In the case of MSCs, function *pnf* assigns to a word $w \in \mathit{pref}(\mathcal{D})$ the minimal word wrt. $<_{\mathrm{lex}}$ that is equivalent to w. To words that are not in $\mathit{pref}(\mathcal{D})$, e.g., an action sequence starting with some receive action, *pnf* assigns an arbitrary receive action from *Act*. Analogously, mapping *snf* assigns to a word $w \in \mathit{suff}(\mathcal{D})$ its normal form, i.e., the minimum (wrt. $<_{\mathrm{lex}}$) among all equivalent words, and it associates with every other word, e.g., an action sequence ending on a sending action, an arbitrary send action from *Act*.

Along the lines of Chapter 5, we define the prefix and the suffix normal form as follows: For $w \in \mathit{Act}^*$, we have:

- If $w \in \mathit{pref}(\mathcal{D})$, then we set $\mathit{pnf}(w) := \min_{<_{\mathrm{lex}}}\{w' \in \mathit{pref}(\mathcal{D}) \mid \exists v \in \mathit{Act}^*\colon wv \approx_{\mathcal{D}} w'v\}$ where $\min_{<_{\mathrm{lex}}}$ returns the minimum of a given set wrt. $<_{\mathrm{lex}}$. Otherwise, let $\mathit{pnf}(w)$ be any arbitrary receive action a (hence, $a \notin \mathit{pref}(\mathcal{D})$).

- If $w \in \mathit{suff}(\mathcal{D})$, then we set $\mathit{snf}(w) := \min_{<_{\mathrm{lex}}}\{w' \in \mathit{suff}(\mathcal{D}) \mid \exists u \in \mathit{Act}^*\colon uw \approx_{\mathcal{D}} uw'\}$. Otherwise, $\mathit{snf}(w)$ is an arbitrary send action a (hence, $a \notin \mathit{suff}(\mathcal{D})$).

Instead of using the lexicographical order $<_{\mathrm{lex}}$ from Chapter 5, we could use so-called *optimal linearizations* introduced in [GKM07] as they provide a very natural normal form in the case of MSCs.

Note that, again, $\mathit{pnf}(\varepsilon) = \mathit{snf}(\varepsilon) = \varepsilon$ and, moreover, $\mathit{pnf}(w) = \mathit{snf}(w)$ iff w is well-formed. As before, the mappings *pnf* and *snf* are canonically extended to sets $L \subseteq \mathit{Act}^*$, i.e., $\mathit{pnf}(L) = \bigcup_{w \in L} \mathit{pnf}(w)$ and $\mathit{snf}(L) = \bigcup_{w \in L} \mathit{snf}(w)$.

As mentioned in Chapter 5, it is crucial for the application of normal forms that a given domain $\mathcal{D}$ satisfies, for all $u, v, u', v' \in \mathit{Act}^*$, properties (*), (**), and (***) (cf. page 75). Under these assumptions, which are satisfied by all the concrete learning setups presented so far, it will indeed be sufficient to look at normal forms when constructing a table in the extension of Angluin's algorithm, which may result in significantly smaller tables. We obtain the extension of EXTENDED-L*, which we call PO-EXTENDED-L* simply by replacing every command of the form $U := U \cup L$ (where L is an arbitrary set of words) by $U := U \cup \mathit{pnf}(L)$, and every command of the form $V := V \cup L$ by $V := V \cup \mathit{snf}(L)$. In particular, EXTENDED-T-UPDATE remains unchanged and is taken from Table 6.2.

The correctness of our improved algorithm is stated in the following corollary.

Corollary 6.3.1. *Let* $(\mathcal{D}, \vdash, \mathit{synth})$ *be a learning setup for class* $\mathfrak{C}$ *of CFMs such that* $\mathcal{D}$ *satisfies properties* (*) *–*(***) *from Chapter 5. Moreover, let* $\mathcal{A} \in \mathfrak{C}$. *If the Teacher classifies/provides MSCs in conformance with* $\mathcal{L}(\mathcal{A})$*, then invoking the new algorithm* PO-EXTENDED-L$^*(\mathcal{D}, \vdash, \mathit{synth})$ *returns, after finitely many steps, a CFM* $\mathcal{A}' \in \mathfrak{C}$ *such that* $L(\mathcal{A}') = L(\mathcal{A})$.

Proof: Consider an instance of (T, U, V) during a run of EXTENDED-L*. For $w \in (U \cup U\mathit{Act})V$, the value of $T(w)$ is $-$ if $w \notin \mathcal{D}$. If, on the other hand, $w \in \mathcal{D}$, then $T(w)$ only depends on the classification of $\mathscr{M}(w)$ by the *Teacher*. So let $u, v \in \mathit{Act}^*$. We consider the two abovementioned cases:

- Suppose $uv \notin \mathcal{D}$. Then, by (*), $u \notin \mathit{pref}(\mathcal{D})$ or $v \notin \mathit{suff}(\mathcal{D})$. Thus, $\mathit{pnf}(u)$ is a receive action or $\mathit{snf}(v)$ is a send action so that $\mathit{pnf}(u) \cdot \mathit{snf}(v) \notin \mathcal{D}$.

- Suppose $uv \in \mathcal{D}$. Then, $u \in \mathit{pref}(\mathcal{D})$ and $v \in \mathit{suff}(\mathcal{D})$. By the definition of the mappings *pnf* and *snf*, there are u' and v' such that $\mathit{pnf}(u) \cdot v' \approx_{\mathcal{D}} uv'$ and $u' \cdot \mathit{snf}(v) \approx_{\mathcal{D}} u'v$. By (**) and (***), $\{\mathit{pnf}(u) \cdot v, u \cdot \mathit{snf}(v)\} \subseteq \mathcal{D}$ so that $\mathit{pnf}(u) \cdot v \approx_{\mathcal{D}} uv$ and $u \cdot \mathit{snf}(v) \approx_{\mathcal{D}} uv$. Applying (**) (or (***)) a second time, we obtain $\mathit{pnf}(u) \cdot \mathit{snf}(v) \in \mathcal{D}$. We deduce $\mathit{pnf}(u) \cdot \mathit{snf}(v) \approx_{\mathcal{D}} uv$, which implies $\mathscr{M}(uv) = \mathscr{M}(\mathit{pnf}(u) \cdot \mathit{snf}(v))$.

Thus, it does not matter if an entry $T(w)$ in the table is made on the basis of w or on $\mathit{pnf}(u) \cdot \mathit{snf}(v)$, regardless of the partitioning uv of w. In particular, if we replace, in U and V, every word with its respective normal form, then the resulting table preserves consistency and closure properties. Moreover, the DFA, that we can construct if the new table is closed and consistent, is isomorphic to that of the original table.

As this replacement is precisely what is systematically done in PO-EXTENDED-L*, the theorem follows. □

Again, the complexity of the modified algorithm depends on the concrete learning setup. Actually, the theoretical time complexity can in general not be improved compared to EXTENDED-L*. However, as Section 9.2 will illustrate by means of several examples, the space complexity can be considerably reduced in the domain of MSCs.

6.4 Related Work

Synthesizing design models or programs from scenarios has received a lot of attention. The goal of this section is to point out and to structure related work.

For scenarios, we distinguish *basic MSCs* from *high-level MSCs* (HMSCs) and *live sequence charts* (LSCs).

Synthesis from Basic MSCs A basic MSC, as used throughout this dissertation, does neither contain loops nor alternatives, and describes a finite set of behaviors. Thus, a finite set of basic MSCs describes a finite set of behaviors. Typically, a system under development has infinitely many behaviors, so that a finite set of scenarios in terms of basic MSCs can only be an approximation of the overall behavior. In fact, the learning algorithm generalizes the finite set of given MSCs to a typically infinite set represented by the design model. In simple words, we synthesize design models from finitely many *examples*. Note also that two different basic MSCs describe different behaviors. Thus, classifying one MSC as desired and one (different) as undesired cannot lead to an *inconsistent* set behaviors. This is in contrast to other approaches which, e.g., employ HMSCs.

One of the first attempts to exploit learning for interactively synthesizing models from examples was proposed in [MS01], where for each process in the system an automaton is inferred using Angluin's learning technique. This procedure is only sensible in the setting of weak CFMs, because the synchronization messages cannot be known before the synthesis procedure. But when you infer product languages by learning local automata and you get, e.g., a counterexample you cannot tell which of the processes does not recognize his projection. It could be one process or several and therefore you cannot relate this counterexample to any process. A severe problem of this approach is that putting the resulting automata in parallel yields a system that may exhibit undesired behavior and may easily deadlock.

Damas et al. use an interactive procedure of classifying positive and negative scenarios for deriving an LTS for each process of the system to be. To this end, they first employ passive learning algorithms RPNI and RPNI^2 (cf. Chapter 3) to infer a global intermediate model that exactly conforms to the given sample but which—as long as the sample does not fulfill certain completeness properties—does not necessarily yield a minimal system model. The global model is subsequently transformed into a distributed system. Then, usually additional effort is required as this projection onto the system's components will normally entail implied behavior such that unwanted "[...] implied scenarios have to be detected and excluded" [DLD05] manually. This is a very complicated and error-prone task, and should in fact not be left to a human being. A further difference to our approach is the communication setting. While Damas et al. work in a synchronous communication environment, which makes all considerations substantially simpler, our approach exhibits the more elaborate setting of completely asynchronous communication, making the target systems larger but, more importantly, more general and closer to real implementations. A minor drawback of using passive learning algorithms might be seen in the extensibility of the hypothesis. If new requirements arise after a model has been generated, the passive learning approach has to be restarted from scratch, whereas applications building on active learning algorithms may benefit from previous models and continue the learning task on basis of the last table inferred.

Another interesting approach using passive learning is delineated in [CW98]. Amongst others, the article describes how to use grammatical inference to derive a formal model of a process from a given stream of system events. In contrast to the previously mentioned approach and our technique, this method only works with positive data, as examples are originating from an event stream obtained from real process executions. Though this procedure may require less user effort than ours, it builds on the restrictive assumption that the events of the process are monitorable by the *Learner*. The authors also shortly comment on detecting concurrency by searching for unrelated events, but leave it for future work to improve their approach.

Similar to [HNS03], [MPO05] propose to use filters, i.e., automated replies to queries, for reducing the number of membership queries that—due to the high number of questions—is usually infeasible for human teachers to answer. The general idea is to exploit additional knowledge of an expert teacher, which, for each negatively answered membership query, specifies prefixes or suffixes for which negative membership is known. In their approach they employ filters for membership and equivalence queries, but conclude that for equivalence queries no substantial improvements are obtained. This result is in contrast to our learning approach which, fortunately, considerably decreases this number.

Synthesis from (High-level) MSCs In contrast to the previously mentioned work, several works consider synthesis from scenarios given in terms of richer formalisms, e.g., MSCs with loops and alternatives, high-level MSCs, or live sequence charts. The underlying assumption is that not only several examples of the expected behavior is given but that the given behavior (mostly) corresponds to the behavior of the system to be. Then, the technical question arising is how to translate from a scenario-based formalism to a state-based formalism. One of the initial works along this line is [KGSB98], which sketches the translation from (high-level) MSCs to statechart models. Similarly, [UKM03] presents a rigorous approach for synthesizing transition systems from high-level MSCs.

The question whether the behavior given by a finite set of MSCs or high-level MSC can in fact be realized by weak CFMs or CFMs is studied, respectively, in [AEY01], [AEY03], and [GMSZ06]. It turns out that the set of scenarios has to meet certain restrictions to be realizable and that the question, whether it is realizable or not, is often undecidable.

Note that describing desired and unwanted behavior in terms of high-level MSCs would allow for inconsistent sets of scenarios as also different high-level MSC may describe a common subset of behaviors.

The works [AEY01, BM07, Gen05, GKM06, GKM07, GMSZ06, HMK$^+$05, Loh03, Mor02] synthesize CFMs from particular classes of finite automata, which can be seen as generalizations of high-level MSCs. Recall that results from [HMK$^+$05, BM07, GKM06, AEY01, Loh03] together constitute Theorem 6.1.14, which, however, is only an ingredient of our algorithms. Our objective is actually to synthesize these particular finite automata that, in turn, allow for constructing a CFM.

In [UBC07], a synthesis technique is proposed that constructs behavior models in the form of Modal Transition Systems (MTS) and is based on a combination of safety properties and scenarios. MTSs distinguish required, possible, and proscribed behavior, and can thus be seen as design models that are more abstract compared to the CFMs that are synthesized using our approach.

Damm and Harel pointed out that the expressiveness of (high-level) MSCs is often inappropriate to specify complete system behavior and introduced the richer notion of live sequence charts (LSCs) [DH01]. Harel's play-in, play-out approach for LSCs [HM03] allows us to execute the possible behavior defined in terms of LSCs, which essentially results in a programming methodology based on LSCs. A similar, executable variant of LSCs, called triggered MSCs, is presented in [SC06].

All the previously mentioned approaches are based on a rather complete, well-elaborated specification of the system to be, such as MSCs with loops or conditions, high-level MSCs, triggered MSCs, or LSCs, whereas for our synthesis approach only basic MSCs have to be provided as examples, simplifying the requirements specification task considerably.

6.5 Summary

To conclude, we summarize the results of learning CFMs in Table 6.3:

- $|\mathcal{A}|$ is the size of the CFM,
- $|Msg|$ is the size of the message alphabet,
- B is the buffer bound (specified in the learning setup), and
- $|Proc|$ is the number of participating processes.

By $poly(.)$, $exp(.)$, and $d\text{-}exp(.)$, respectivcely, we denote polynomial, exponential, and doubly exponential complexity, respectively, in the parameters given.

CFM class	#equivalence queries	#membership queries
deterministic $\forall B$-bounded	$(\lvert\mathcal{A}\rvert \cdot \lvert Msg\rvert + 1)^{B\cdot\lvert Proc\rvert^2+\lvert Proc\rvert}$	$poly(\lvert\mathcal{A}\rvert, \lvert Msg\rvert)$ $exp(B, \lvert Proc\rvert)$
$\forall B$-bounded deadlock-free	$2^{(\lvert\mathcal{A}\rvert \cdot \lvert Msg\rvert + 1)^{B\cdot\lvert Proc\rvert^2+\lvert Proc\rvert}}$	$exp(\lvert\mathcal{A}\rvert, \lvert Msg\rvert)$ $d\text{-}exp(B, \lvert Proc\rvert)$
$\exists B$-bounded	$2^{(\lvert\mathcal{A}\rvert \cdot \lvert Msg\rvert + 1)^{B\cdot\lvert Proc\rvert^2+\lvert Proc\rvert}}$	$exp(\lvert\mathcal{A}\rvert, \lvert Msg\rvert)$ $d\text{-}exp(B, \lvert Proc\rvert)$
deterministic $\forall B$-bounded deadlock-free weak	$(\lvert\mathcal{A}\rvert \cdot \lvert Msg\rvert + 1)^{B\cdot\lvert Proc\rvert^2}$	$poly(\lvert\mathcal{A}\rvert)$ $exp(\lvert\mathcal{A}\rvert)$ $d\text{-}exp(B, \lvert Proc\rvert)$

Table 6.3: An overview over the query complexity of learning CFMs

Part II

Software Development Employing Learning Techniques

7 The `Smyle` Modeling Approach

The following chapter deals with a new software engineering lifecycle model, called the `Smyle` *Modeling Approach*, or `SMA` for short, which is based on the learning approach for distributed systems described in the previous chapter.

Within the `SMA`, we want to employ the idea of *Model Driven Development* (MDD, for short) [TBD07, MCF03] and thereby opt for short development cycles and cost reduction. Other advantages of the MDD are:

(i) *"timeless" models*, which are reusable at any time because the abstract model, i.e., the formal description of the system to be (e.g., in terms of MSCs, DFA, or CFMs), stays available and can be adapted when necessary and

(ii) *platform independence* of the derived model (e.g., the CFM in our case) as a consistent description of the final system. This makes later changes of the underlying architecture of the final model possible.

The general idea of MDD is that an abstract model of the system to be might be a better documentation of the whole project than the code itself. Moreover, in MDD the abstract model is refined or transformed in several stages from an initial set of requirements specified in natural language over several abstract models to a final implementation. This fits very well in our incremental learning setting.

The rest of this chapter is organized as follows: after a preliminary section in which we become acquainted with a specification formalism for easing the software engineers task within the `SMA`, we go into details about the `SMA` by presenting a global view first, and then highlight the internal processes of our approach. Subsequently, we compare our lifecycle model with well-known and well-established lifecycle models. Finally, we close this chapter by giving an example for deriving a small design model.

7.1 Preliminaries

In this section, we want to introduce a formalism which will subsequently be employed within the `SMA` to ease a designer's task of classifying scenarios as either wanted or undesired. The goal is to provide means for specifying template-like *patterns* subsuming collections of MSCs using a logic called PDL [BKM07]. In the following, we will first detail on the motivation for choosing PDL, subsequently describe its syntax and semantics, and, finally, present several formula instances to exemplify its use.

MSC Patterns We choose the logic PDL for three reasons:

(i) It is expressive, subsuming, e.g., Peled's temporal logic TLC$^-$ [Pel00].

(ii) It combines easy to understand and engineer-friendly concepts such as regular expressions and boolean operators.

(iii) Its membership problem, i.e., to decide if a given MSC satisfies a given formula, can be solved in polynomial time, as we will show at the end of this section.

Note that PDL is a quite expressive logic and that satisfiability for PDL is undecidable [BKM07]. Fortunately, the `SMA` only builds on deciding membership problems, as they allow us to determine if a given scenario belongs to a property that represents good or undesired behavior. This is the principal reason why PDL is better suited for our purposes than, say, high-level MSCs, whose membership problem is NP-complete [AEY01].

The building blocks of formulae in our logic PDL are regular expressions and boolean connectives, which can be nested arbitrarily. Let us first describe the syntax of PDL.

Definition 7.1.1 (PDL Syntax). *The actual syntax of PDL is described by the following rules where φ describes* local *formulae, α defines regular expressions, called* path *expressions, and Φ specifies PDL formulae:*

$$\begin{array}{lcll}
\varphi & ::= & \mathbf{true} \mid \sigma \mid \neg\varphi \mid \varphi \vee \varphi \mid <\alpha>\varphi \mid <\alpha>^{-1}\varphi & \textit{(local formulae)} \\
\alpha & ::= & \{\varphi\} \mid \mathsf{proc} \mid \mathsf{msg} \mid \alpha;\ \alpha \mid \alpha + \alpha \mid \alpha^{*} & \textit{(path expressions)} \\
\Phi & ::= & \mathbf{E}\varphi \mid \neg\Phi \mid \Phi \vee \Phi & \textit{(PDL formulae)}
\end{array}$$

In this definition $\sigma \in Act$ represents a communication action (e.g., $!(1,2,a)$). Besides these basic modalities, we use common shorthands such as $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ for $\neg\varphi_1 \vee \varphi_2$, **false** *for $\neg$***true***, $\sqsubset\alpha\sqsupset\varphi$ for $\neg<\alpha>\neg\varphi$, $\sqsubset\alpha\sqsupset^{-1}\varphi$ for $\neg<\alpha>^{-1}\neg\varphi$, and $\mathbf{A}\Phi$ for $\neg\mathbf{E}\neg\Phi$.*

In the following we will describe the formal semantics of PDL and, subsequently, exemplify its use within the learning setting.

Definition 7.1.2 (PDL Semantics). *Let $M = (E, \preceq, \lambda)$ be an MSC (cf. Definition 6.1.1), and $e \in E$ a communication event:*

Local formulae:

$$\begin{array}{lll}
M, e \models \mathbf{true} & \textit{for all } e \in E & \\
M, e \models \sigma & \Longleftrightarrow & \lambda(e) = \sigma \quad \textit{for } \sigma \in Act \\
M, e \models \neg\varphi & \Longleftrightarrow & M, e \not\models \varphi \\
M, e \models \varphi_1 \vee \varphi_2 & \Longleftrightarrow & M, e \models \varphi_1 \quad \textit{or} \quad M, e \models \varphi_2
\end{array}$$

Local formulae with path expressions:

forward:

$$\begin{array}{lll}
M, e \models <\{\psi\}>\varphi & \Longleftrightarrow & M, e \models \psi \quad \textit{and} \quad M, e \models \varphi \\
M, e \models <\mathsf{proc}>\varphi & \Longleftrightarrow & \exists\, p \in \textit{Proc}\,,\ e' \in E : (e, e') \in \lessdot_p \textit{ and} \quad M, e' \models \varphi \\
M, e \models <\mathsf{msg}>\varphi & \Longleftrightarrow & \exists\, e' \in E : (e, e') \in \prec_{\mathrm{msg}} \textit{ and } M, e' \models \varphi \\
M, e \models <\alpha_1;\ \alpha_2>\varphi & \Longleftrightarrow & M, e \models <\alpha_1><\alpha_2>\varphi \\
M, e \models <\alpha_1 + \alpha_2>\varphi & \Longleftrightarrow & M, e \models <\alpha_1>\varphi \vee <\alpha_2>\varphi \\
M, e \models <\alpha^{*}>\varphi & \Longleftrightarrow & \exists\, n \in \mathbb{N} : M, e \models (<\alpha>)^{n}\varphi
\end{array}$$

backward:

$$\begin{array}{lll}
M, e \models <\{\psi\}>^{-1}\varphi & \Longleftrightarrow & M, e \models \psi \quad \textit{and} \quad M, e \models \varphi \\
M, e \models <\mathsf{proc}>^{-1}\varphi & \Longleftrightarrow & \exists\, p \in \textit{Proc}\,,\ e' \in E : (e', e) \in \lessdot_p \textit{ and } M, e' \models \varphi \\
M, e \models <\mathsf{msg}>^{-1}\varphi & \Longleftrightarrow & \exists\, e' \in E : (e', e) \in \prec_{\mathrm{msg}} \textit{ and } M, e' \models \varphi \\
M, e \models <\alpha_1;\ \alpha_2>^{-1}\varphi & \Longleftrightarrow & M, e \models <\alpha_1>^{-1}<\alpha_2>^{-1}\varphi \\
M, e \models <\alpha_1 + \alpha_2>^{-1}\varphi & \Longleftrightarrow & M, e \models <\alpha_1>^{-1}\varphi \vee <\alpha_2>^{-1}\varphi \\
M, e \models <\alpha^{*}>^{-1}\varphi & \Longleftrightarrow & \exists\, n \in \mathbb{N} : M, e \models (<\alpha>^{-1})^{n}\varphi
\end{array}$$

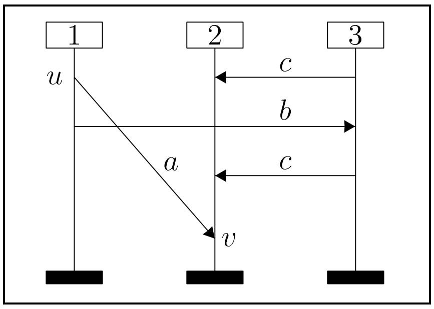


Figure 7.1: An MSC

PDL formulae:

$$\begin{array}{lcl} M \models \mathbf{E}\varphi & \iff & \exists\, e \in E : M, e \models \varphi \\ M \models \neg\Phi & \iff & M \not\models \Phi \\ M \models \Phi_1 \vee \Phi_2 & \iff & M \models \Phi_1 \text{ or } M \models \Phi_2 \end{array}$$

Let us describe the semantics more intuitively: First, we let $M, e \models \varphi$ iff e is labeled with φ. The boolean connectives are as usual. Moreover, $M, e \models <\alpha>\varphi$ iff there is an event e' such that $M, (e, e') \models \alpha$ and $M, e' \models \varphi$. The interpretation of the inverse operator $<\alpha>^{-1}$ is, in a sense, dual; we just replace the condition $M, (e, e') \models \alpha$ with $M, (e', e) \models \alpha$. The intuition is that a path expression α defines a binary relation of events: For an MSC M and two of its events u and v, we write $M, (e, e') \models \mathsf{proc}$ iff e and e' are connected in M by a process edge (where e needs to be the successor of e'), i.e., there exists a process $p \in \mathit{Proc}$ such that $(e, e') \in \lessdot_p$. Accordingly, the relation $M, (e, e') \models \mathsf{msg}$ holds iff e and e' are connected by a message arrow, i.e., $(e, e') \in \prec_{\mathrm{msg}}$. The composition $\alpha_1;\ \alpha_2$ defines the set of pairs (e, e'), for which there exists an event e'' such that (e, e'') is in the semantics of α_1 and (e'', e') is in the semantics of α_2. The formula α^* describes the reflexive, transitive closure of the relation defined by α, and $\alpha_1 + \alpha_2$ defines the union of the relations induced by α_1 and α_2. Finally, the semantics of $\{\varphi\}$ is the set of pairs (e, e) such that $M, e \models \varphi$. Hereby, φ is a local formula, which is interpreted as described above. The PDL formulae are interpreted over MSCs. The only formula that requires explanation is $\mathbf{E}\varphi$. It is satisfied by MSC M iff there is an event e of M such that $M, e \models \varphi$. Universal quantification (e.g., $M \models \mathbf{A}\varphi$) and conjunctions can, as usually, be formalized using negation and existential quantification or disjunction, respectively. The subformula φ in $\mathbf{E}\varphi$ or $\mathbf{A}\varphi$ is thus interpreted at events of an MSC. It might be of the form $<\psi>\varphi'$ meaning that, starting in the event under consideration, there is a ψ-labeled path to another event that satisfies φ'. The dual construct $\sqsubset\psi\sqsupset\varphi'$ expresses that the property φ' has to hold at any event that can be reached following a ψ-labeled path. In the following, let us write φ for the path expression $\{\varphi\}$ to simplify presentation.

Having formally and intuitively introduced the syntax and semantics of PDL, we now will provide some example formulae demonstrating the expressive power of PDL, as well as its usage within the `SMA`.

Example 7.1.3 (PDL Formulae)**.** To be able to grasp the expressive power of PDL and to obtain a better feeling of how to use this logic for specifying patterns in the setting of the `SMA`, we now consider several PDL formulae and describe their intuitive meaning. A typical local formula is $!(1, 2, a)$. It is valid in all nodes whose labeling is $!(1,2,a)$, e.g., in event u of Figure 7.1. The path expression $(!(1,2,a);\ \mathsf{proc})^*$ pairs events e and e' such that e and e' are located on the same process line and, on the very process line, any event in between e and e' is labeled with send action $!(1,2,a)$. Given local formulae φ and ψ, the local formula $<(\{\varphi\}\ ;\ (\mathsf{proc} + \mathsf{msg}))^*>\psi$ corresponds to the "until" construct $\varphi\mathcal{U}\psi$ in Peled's TLC^-. I.e., for MSC M and an event u of M, $M, u \models <(\{\varphi\}\ ;\ (\mathsf{proc} + \mathsf{msg}))^*>\psi$ iff there is, roughly speaking, a (forward) path from e to some e' with $M, e' \models \psi$ along which φ is permanently valid (however, we do not require $M, e' \models \varphi$). Note that, in TLC^-, one cannot express that an MSC contains an even number of messages, which is, however, definable in PDL . To ease the presentation, let us assume that we have an action alphabet where we abstract from message contents, i.e., $Act = \{!_p, ?_p, !_q, ?_q\}$ over two processes p, q, and let us, moreover, suppose that we only want to count the sending events along process p. The PDL formula stating that an MSC must have an even number of messages being sent from process p then is: $\mathbf{A}\,(\mathbf{procmin}{\rightarrow}{<}((?_q;\ \mathsf{proc})^*;\ !_p;\ \mathsf{proc};\ (?_q;\ \mathsf{proc})^*;\ !_p;\ \mathsf{proc};\ (?_q;\ \mathsf{proc})^*)^*{>}\ \mathbf{procmax})$. Note that we used a short notation and wrote $!_p$ and $?_q$ instead of $\{!_p\}$ and $\{?_q\}$. The local formulae $\mathbf{procmin} := \sqsubset\mathsf{proc}\sqsupset^{-1}\mathbf{false}$ and $\mathbf{procmax} := \sqsubset\mathsf{proc}\sqsupset\ \mathbf{false}$ express that an event is the first or the last on its process line, respectively. Now consider M to be the MSC from Figure 7.1 with designated nodes u and v. We have:

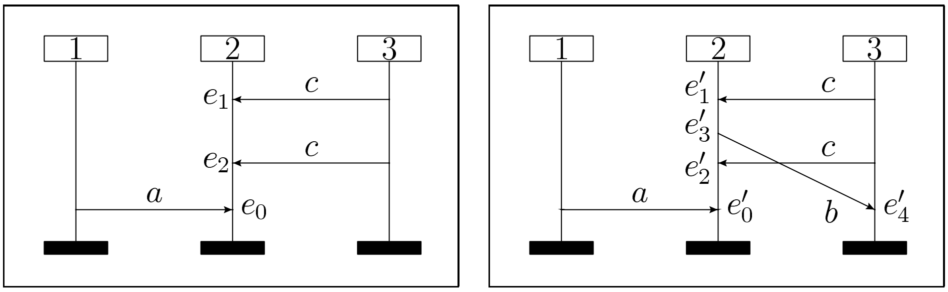
1) $M, u \models \mathbf{procmin}$
2) $M, u \models <(\mathsf{proc} + \mathsf{msg})^*>v$ (or equivalently: $M, (u, v) \models <(\mathsf{proc} + \mathsf{msg})^*>$)

The first item examines a local formula saying that "event u is minimal on its process". Considering Figure 7.1 this statement is valid, however u is not the smallest event in M because there is another minimal event $!(3,2,c)$ on process 3. The local formula in item 2) states a simple property, namely "event u happens before event v" which is the case for the MSC from Figure 7.1 as well.

We now turn to global formulae, i.e., we give some examples for PDL formulae (again using the shorter version omitting the curly braces around local formulae):

$$\begin{aligned}
\varphi_1 &= \mathbf{A}\big(<(\mathsf{proc} + \mathsf{msg})^*>(\mathbf{procmax} \wedge ?(2,1,a))\big),\\
\varphi_2 &= \mathbf{E}\big(\mathbf{procmax} \wedge ?(2,1,a)\big),\\
\varphi_3 &= \mathbf{A}\big(\sqsubset ?(2,3,c);\ \mathsf{proc};\ ?(2,3,c) \sqsupset \mathbf{false}\big),\ \text{and}\\
\varphi_4 &= \mathbf{A}\,(\mathbf{procmin}{\rightarrow}{<}((?_q;\mathsf{proc})^*;!_p;\mathsf{proc};(?_q;\mathsf{proc})^*;!_p;\mathsf{proc};(?_q;\mathsf{proc})^*)^*{>}\mathbf{procmax})\,.
\end{aligned}$$

The universal formula φ_1 describes that, from any event, there is an arbitrarily labeled path through the MSC (expressed by the first part of the local formula: $<(\mathsf{proc} + \mathsf{msg})^*>$) to another event that is maximal on its process (expressed by the local formula **procmax**) and labeled with the receive event $?(2,1,a)$. In other words, there must be a greatest event in the MSC at hand and this greatest event shall be labeled with $?(2,1,a)$. Indeed, φ_1 is satisfied by the MSC M_1 from Figure 7.2(a) because the greatest and maximal event is e_0. The MSC M_2 from Figure 7.2(b), however, does not satisfy φ_1 as there are two maximal events e'_0 and e'_4 and, hence, no greatest. Note that the existential formula φ_2 is not equivalent to φ_1, as φ_2 only requires the existence of an event that is *maximal* on process 2; but this event needs not be the greatest in the MSC. As a matter of fact, both MSCs satisfy φ_2 because both contain at least one maximal event labeled by receive action $?(2,1,a)$. The universal formula φ_3 forbids two consecutive events both labeled with $?(2,3,c)$. It thus is refuted by MSC M_1 because $(e_1, e_2) \in \preceq_2$ but accepted by MSC



(a) MSC M_1 (b) MSC M_2

Figure 7.2: Sample MSCs

M_2 because there exists an event $e'_3 = !(2,3,b)$ in between the events e_1 and e_2 labeled with $?(2,3,c)$. Formula φ_4 was already introduced before. It characterizes all MSCs (over two processes p, q) that have an even number of messages being sent from p. If one wanted to extend this formula to a larger process set $Proc'$ and action alphabet Act', one had to replace the simplified actions $!p, ?_q$ by disjunctions over $Proc'$ and Act'. ◇

The examples above exemplarily show the power of PDL. It becomes clear, that employing PDL, MSCs can be automatically classified according to the patterns specified, yielding a substantial decrease of user queries and, hence, an enormous relief for the designer acting as a teacher of the learning algorithm embedded in the `SMA`. It remains to show that this method can be used efficiently.

Essential for our setting using `Smyle` within the `SMA` is the result that we can efficiently check whether a given MSC adheres to a given PDL formula, which can be proven as follows:

Theorem 7.1.4. *The membership problem for* PDL *is in* PTIME, *even if the number of processes is part of the input. More precisely, given a* PDL *formula φ and an MSC M, we can decide in time $\mathcal{O}(|M| \cdot |\varphi|^2)$ if $M \models \varphi$, where $|M|$ denotes the number of events in M and $|\varphi|$ denotes the length of φ.*

Proof: [sketch] Let Φ be the given PDL formula. In subformulae $<\alpha>\varphi$ and $<\alpha>^{-1}\varphi$ of Φ, we consider α as regular expression over some finite alphabet $\{\mathsf{proc}, \mathsf{msg}, \{\varphi_1\}, \ldots, \{\varphi_n\}\}$ with local formulae $\varphi_1, \ldots, \varphi_n$. Any such expression can be transformed into a corresponding finite automaton of linear size. We proceed by inductively labeling events of the given MSC with states of the finite automata. This state information is then used to discover whether or not an event of M satisfies a subformula $<\alpha>\varphi$ or $<\alpha>^{-1}\varphi$, which yields labelings in $\{0,1\}$. Boolean combinations and $\mathbf{E}\varphi$ are then handled in a straightforward manner.

An algorithm solving the PDL membership problem is given in Tables C.1, C.2 and C.3 in Appendix C. □

Within the `SMA`, the logic will be employed as follows: positive and negative sets of formulae Φ^+ and Φ^- are input by the user, via an integrated formula editor. An example for a negative statement would be, say, "*there are two receives of the same message in a row*", which corresponds to the negation of the PDL formula φ_3 above. An annotated MSC for this example formula is given in Figure 7.9 (c) on page 128. Then, the learning algorithm can autonomously and efficiently check for all formulae $\varphi^+ \in \Phi^+$, $\varphi^- \in \Phi^-$ and unclassified MSCs M whether $M \not\models \varphi^+$ or $M \models \varphi^-$. If one of the two cases occurs, then the set of negative samples is updated to $\{M\} \cup \mathbb{M}^-$, where $\mathbb{MSC}$ contains all MSCs classified as negative so far. In all other cases the question is passed to the user.

7.2 The `SMA` in Detail

Let us now concentrate on the `SMA` in detail. It is common knowledge [Eas04] that traditional engineering lifecycle models like the well-known *Waterfall model* [Roy87, Som06, Pre04, GJM02] or the V-model [Pre04, Som06] suffer from some severe deficiencies, despite their wide use in today's software development. One of the problems is that both models assume implicitly that a complete set of requirements can indeed be formulated at the beginning of the software engineering lifecycle. Although in both approaches it is possible to revisit a previously passed phase, this is considered a backwards step involving time-consuming reformulation of documents, models, or code produced in the previous and current phases, which, in turn, cause high additional costs for redesign.

The nature of a typical software engineering project is, however, that requirements are usually incomplete, often contradicting, and frequently changing during the project evolution. A high-level design, on the other hand, is typically a complete, and consistent model that is expected to conform to the requirements. Thus, especially the step from requirements to a high-level design is a major challenge within a software engineering lifecycle: The incomplete set of requirements has to be made complete and inconsistencies have to be eliminated. An impressive example for inconsistencies in industrial-size applications is given by Holzmann [Hol94] where for the design and implementation of a part of Signaling System 7 in the 5ESS®switching system (the ISDN User-Part protocol defined by the CCITT) "almost 55% of all requirements from the original design requirements [...] were proven to be logically inconsistent [...]".

Moreover, also later stages of the development process often require additional modifications of requirements and the corresponding high-level design, either due to changing user requirements or due to unforeseen technical difficulties. Thus, besides the step of generating a complete and consistent set of requirements and a conforming design model in the initial stages of a development process, a lifecycle model should support an easy adaptation of requirements and its conforming design model also at later stages. The `SMA` is a new software engineering lifecycle model that addresses these goals.

However, as we discuss later, it may also be employed to enrich existing lifecycle models.

7.2.1 A Bird's-eye View on `SMA`

The `Smyle` *Modeling Approach* is a software engineering lifecycle model tailored to distributed systems. A prerequisite is that the participating processes and their communication actions can be fixed in the first steps of the development process, before actually deriving a design model. Requirements for the behavior of the involved processes, however, may be given vaguely and incomplete first but are made precise within the process. While clearly not every development project fits these needs, a considerable amount of systems especially in the automotive domain do, which motivated to develop the `SMA`.

Within `SMA`, our goal is to round-off requirements, remove inconsistencies and to provide methods catering for modifications of requirements in later stages of the software engineering lifecycle. One of the main challenges to achieve these goals is to come up with simple means for concretizing and completing requirements as well as resolving conflicts in requirements. We attack this intrinsically hard problem using the following rationale:

> While it is hard to come up with a complete and consistent formal specification of the requirements, it is feasible to classify exemplifying behavior as desired or illegal. (`SMA` rationale)

This rationale builds on the well-known experience that human beings prefer to explain, discuss, and argue in terms of example scenarios, but are often over-strained when having to give precise and universally valid definitions. Thus, while the general idea to formalize requirements, for example using temporal logic, is in general desirable, this formalization is often too cumbersome and, therefore, not cost-effective—and the result is, unfortunately, often too error-prone. This is because it is hard to have a clear (i.e., complete and consistent) picture of the system to develop right at the beginning of the software engineering lifecycle.

This also justifies our restriction to MSCs without branching, if-then-else, and loops, when learning design models: It may be too error-prone to classify complex MSCs (or equally or even more expressive notions like HMSCs or LSCs) as either wanted or unwanted behavior.

Our experience with requirements documents shows that especially requirements formulated in natural language are often explained in terms of scenarios, showing wanted or unwanted behavior of the system to develop. Additionally, it is evident that it is easier for the customer to judge whether a given simple scenario is intended or not, in comparison to answering whether a formal specification matches the customer's needs.

The key idea of the `SMA` is therefore to incorporate the novel learning algorithm called EXTENDED-L* (cf. Chapter 6) with supporting tool `Smyle` (cf. Chapter 9) for synthesizing design models based on scenarios explaining requirements. Thus, requirements- and high-level design phase are interweaved. `Smyle`'s nature is to extend initially given scenarios to consider, for example, corner cases: It *generates* new scenarios whose classification as desired or undesired is indispensable to complete the design model and asks the engineer exactly these scenarios. Thus, the learning algorithm actually causes a natural iteration of the requirements elicitation and design model construction phase. Note that `Smyle` synthesizes a design model that is indeed consistent with the given scenarios and thus does precisely exhibit the scenario behavior, and nothing more. That is in contrast to the existing learning algorithms like [MS01, UKM03, DLD05], mentioned in the previous chapter.

`SMA` is tailored to component-based systems communicating with each other to achieve a common goal. A natural design model for such systems are CFMs (cf. Definition 6.1.2 on page 87). Exemplifying behavior (*scenarios*) of such systems is best given in terms of MSCs (cf. Definition 6.1.1 on page 84). Thus, `SMA`, similar as its learning algorithm `Smyle`, is designed to derive CFMs based on either positively or negatively classified MSCs, representing wanted or, respectively, unwanted behavior of the software system to build.

While `SMA`'s initial objective is to elaborate on the inherent correspondence of requirements and design models by asking for further exemplifying scenarios, it also provides simple means for modifications of requirements later in the design process. Whenever, for example in the testing phase, a mismatch of the implementation's behavior and the design model is witnessed, which can be traced back to an invalid design model, it can be formulated as a negative scenario and can be given to the learning algorithm to update the current design model. This will, possibly after considering further scenarios, modify the design model to disallow the unwanted behavior. Thus, necessary modifications of the current software system in later phases of the software engineering lifecycle can easily be fed back to update the design model. This high level of automation is aiming at an important reduction of development costs.

7.2.2 The `SMA` Lifecycle Model

The `Smyle` Modeling Approach, cf. Figure 7.3, consists of a requirements phase, a high-level design phase, a low-level design phase, and a testing and integration phase. Following modern *model-based* design lifecycle models, the implementation model is transformed automatically into executable code, as it is increasingly done in the automotive and avionics domain.

In the following, the main steps of the `SMA` lifecycle model are described in more detail, with a focus on the phases depicted in Figure 7.3 and a brief discussion on testing and integration phases.

Derivation of a Design Model

According to Figure 7.3, the derivation of design models is divided into three steps:

The first phase is called *scenario extraction phase*. Based on the usually incomplete system specification the designer has to infer a set of scenarios which will be used as input to `Smyle`. It is worthwhile to study the results from [Kof07, Kof08] in this context, which allow us to infer MSCs from requirements documents by means of natural language processing tools, potentially yielding (premature) initial behavior. After collecting this initial set of MSCs representing desired and undesired system behavior, the second phase initiates the learning algorithm to learn a system model based on these scenarios.

In the *learning and simulation phase*, the designer and client (referred to as *stakeholders* in the following) will work hand in hand according to the *designing-in-pairs* paradigm. The advantage is that both specific knowledge about requirements (contributed by the customer) and solutions to abstract design questions (contributed by the designer) coalesce into one model. With its progressive nature, `Smyle` attempts to derive a model by interactively presenting new scenarios to the stakeholders which in turn have to classify them as either positive or negative system behavior. Due to the evolution of requirements implied by this categorization, the requirements document should automatically be updated incorporating the new MSCs. Additionally, the most important scenarios are to be user-annotated with the reason for the particular classification to complement the documentation. When the internal model is complete and consistent with regard to the scenarios classified by the stakeholders, the learning procedure halts, and `Smyle` presents a frame for simulating and analyzing the current system. In this dedicated simulation component—depicted in Figure 7.8(a) and (c) on page 128—the designer and customer pursue their designing-in-pairs task and try to obtain a first impression on the system to be by executing events and monitoring the resulting system behavior depicted as an MSC. In case missing requirements are detected, the simulator can extract a set of counterexample MSCs, which should again be augmented by the stakeholders to complete documentation. These MSCs are then introduced to `Smyle`, whereupon the learning procedure continues until reaching the next consistent automaton.

The designer then advances to the *synthesis and analysis phase* where a distributed model (a CFM) is synthesized in an automated way. To get diagnostic feedback as soon as possible in the software engineering lifecycle, a subsequent analysis phase asks for an

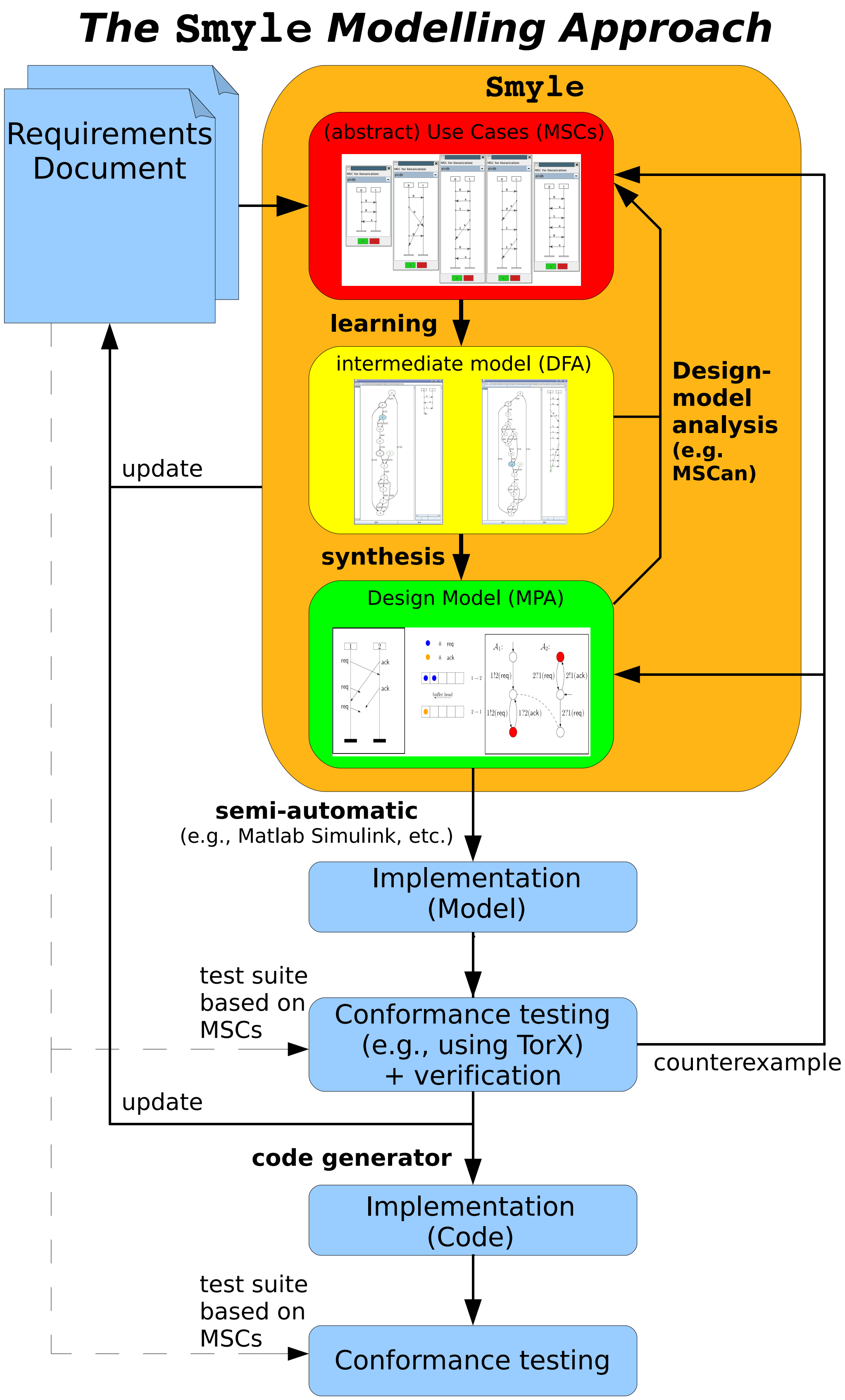

The Smyle Modelling Approach
Requirements Document
Smyle
(abstract) Use Cases (MSCs)
learning
intermediate model (DFA)
update
Design-model analysis (e.g. MSCan)
synthesis
Design Model (MPA)
semi-automatic
(e.g., Matlab Simulink, etc.)
Implementation (Model)
test suite based on MSCs
Conformance testing (e.g., using TorX) + verification
counterexample
update
code generator
Implementation (Code)
test suite based on MSCs
Conformance testing

Figure 7.3: SMA: The Smyle Modeling Approach

intensive analysis of the current design model. Consulting model-checking-like tools[1] as `MSCan` [BKSS06], which are designed for checking dedicated properties of communicating systems, might lead to additional knowledge about the current model and its implementability. With `MSCan` the designer is able to check for potential deficiencies of the forthcoming implementation, like *non-local choice* [BAL97] or *non-regularity* [HMK$^+$05], i.e., process divergence, etc. The counterexamples generated by `MSCan` are again MSCs and as such can be smoothly fed back to the learning phase. Instead of employing tools, the engineer could of course also try to alter the distributed model herself. Nevertheless, this is not encouraged as it obviously violates and breaks the learning-based lifecycle, and conceals the danger of accidentally adding or deleting system behavior because distributed systems are easily misunderstood by humans. If the customer and designer are satisfied with the result the client's presence is not required anymore and their direct collaboration terminates. Note that the design model obtained at this stage may also serve for a legal contract describing the system to be built.

Enhancing the Learning Process

While it is hard to come up with a universally valid specification right in the beginning of the design phase, typical *patterns* of clearly allowed or disallowed MSCs usually are observed during the learning phase. Hence, when applying the `SMA`, it is useful to provide expressive though concise means to describe MSC languages, e.g., to specify mandatory or unwanted system behavior. Over words, temporal logics such as LTL have emerged as an important ingredient in the verification and synthesis of reactive systems. For MSCs, only few attempts to define a suitable temporal logic exist. The lack of temporal logics probably traces back to the complexity of MSCs: even simple temporal logics over MSCs have an undecidable satisfiability problem. For `SMA`, we adopt the logic PDL proposed in [BKM07] and introduced in the previous section, which is inspired by the propositional dynamic logic by Fischer and Ladner [FL79], but adapted to MSCs.

The purpose of employing PDL within the `SMA` is to decrease the number of MSCs to classify, for considerably reducing the designer's efforts. If the desirable or undesirable properties obey a certain structure these so-called *patterns*, representing temporal-logic properties, should be expressed in PDL directly or, ideally, marked—as in Figure 7.9—within an MSC featuring this behavior. In case patterns were marked in the MSCs they should (semi-)automatically be transformed into PDL formulae. Afterwards, they have to be categorized as either positive or negative, as in the case of classifying MSCs. An unclassified MSC has to fulfill all positive patterns and must not fulfill any negative pattern in order to be passed to the designer. In case any positive pattern is not fulfilled or any negative pattern is fulfilled the scenario can be classified as negative without user interaction. Roughly speaking: *employing a set of formulae in the learning procedure will further ease the designers task* because she has to classify less scenarios as many of them can be answered by the specified patterns.

Transformation to an Implementation Model

The engineer's task now is to semi-automatically transform the design model into an implementation model. Of course, as no software lifecycle can claim to be a universal remedy also `SMA` requires manual effort and human ingenuity. For this purpose the `SMA` proposes to employ tools like *Matlab Simulink* which takes as input for example a so-called *Stateflow diagram* [HR04] and transforms it into an implementation model. Hence, the manual effort the designer has to perform in the current phase reduces to transforming the CFM (as artifact of the design phase) into the input language (e.g., Stateflow) of the tool used for deriving the implementation model. Another possibility is to employ the approach described in Balarin et al. [BCG$^+$99] where `C` code is synthesized from *cooperating finite-state machines* (CFSMs), a communicating automaton model related to the CFM model used in this dissertation. The manual effort in this case consisted in transforming the CFM as artifact of our learning process into a CFSM as input for the synthesis process by Balarin et al. As the authors state, this method is only applicable to a restricted class of embedded systems for which features like loop bounds, which are determined at runtime, recursion, etc., must not be used. As these limitations are rather severe, this method can, in general, not be applied to all kinds of embedded software. Note that Balarin et al. use CFSMs as high-level representation and input to their synthesis method. The authors, however, do not state how to obtain these communicating automata. Thus, the `SMA` approach could be of interest for their setting, too, because it would allow for correct CFM derivation.

---

[1] Note that currently there are no general purpose model checkers for CFMs available.

To sum up: depending on the complexity and requirements of the system to be either of the above procedures should be performed in order to evolve to the next phase of the software development cycle.

Conformance Testing

As early as possible the implementation model should pass a testing phase before being transformed into real code to lower the risk of severe design errors and supplementary costs. `SMA` employs *model-based testing* [BJK$^+$05] as it allows a much more systematic treatment by mechanizing the generation of tests as well as the test execution phase.

Within the `SMA`, we now have reached a point at which it becomes necessary to employ such techniques. Let us briefly explain the main goal of conformance testing.

In general, this testing technique is employed to check the conformance of given standards (here represented by the requirements document or the abstract model, respectively) with regard to the implementation model or implementation. Conformance testing (or model-based testing) contains the following steps: first an abstract model of the system has to be derived. In our context this model is inferred using learning techniques presented in Chapter 6 yielding an abstract model of the specification. From this abstract model we can then, in a second step, generate (abstract) test cases and finally in the third step test the (partially) manually coded implementation parts.

As mentioned previously, the output of the design model derivation phase is an abstract model of the communication behavior. It completely and correctly describes the communication structure but compared to a real implementation lacks, e.g., code for memory management, file handling, network communication, etc.

Let us, for example, consider a web server. The model of the communication behavior of this application is learned using our tool `Smyle` (cf. Chapter 9). The model abstractly describes all interactions of the webserver with a client (e.g., the login to a password-protected website, or the exchange of documents). On this level of abstraction we want to perform our tests wrt. the implementation model. To this end, we can almost directly use the MSCs drawn or derived (using the approaches like [Kof07]) from the requirements document and apply them to the implementation. Because the communication behavior can be expected to be correct by construction it does not have to be tested. This is, because usually we assume that the user classifies MSCs correctly, and code generators generate correct output. Thus, as long as the system under development is not safety-critical or has to fulfill high security standards, one can refrain from testing the automatically generated code fragments. The semi-automatically derived implementation model and the manually coded rest of the implementation (referred to in the following paragraph), however, can contain errors. A thorough testing phase is therefore indispensable and mandatory.

The conformance testing phase can be summarized as follows. We want to detect the bugs employing conformance testing using as a natural test suite (i.e., a set of tests) the MSCs from the requirements document, i.e., the MSCs originally contained in this document, augmented by the MSCs that were classified during the learning phase. Additionally, we can extend this test suite by employing our test case generation approach `Style` which will be described at the end of Chapter 9. If the designer detects a failure during the testing phase, counterexamples in form of abstract system runs and, thus, MSCs are automatically generated, and again the requirements document is updated accordingly, enclosing the new scenarios and their corresponding requirements derived by the designer. At last, the generated scenarios are introduced into `Smyle` to derive the next model.

In practice, model-based testing has been implemented in several software tools and has demonstrated its power in various case studies [CSH03, BJK$^+$05]. For the testing phase, the `SMA` recommends tools like *TorX* or *TGV* [BFS04].

Synthesis of Code, Testing, and Maintenance

Having converged to a final, consistent implementation model, a code generator is employed for generating code skeletons or even entire code fragments for the distributed system. These fragments then have to be completed by programmers such that, afterwards, the software can finally be installed at the client's site.

As stated in the previous paragraph, as soon, as human beings contribute real code to the implementation, conformance testing should be stipulated. As in the current phase the automatically derived code skeletons were augmented by hand-written code, we schedule a second phase of testing. Similar to the last phase, we can now use MSCs from the requirements document, or automatically generated ones (using, e.g., `Style`), to perform parts of the tests on an abstract level in order to receive diagnostic feedback. Of course, in this phase of the software development also concrete test cases have to be created. To this end, however, the abstract tests can serve as templates to implement real test cases.

Concerning the extensibility of the system, we obtain the following result: if new requirements arise after some operating time of the system, the old design model can be upgraded by resurrecting the `SMA` on basis of the already classified, or even partially reclassified set of scenarios, learning the new model, synthesizing the new system as explained and, thus, closing the `SMA` lifecycle.

7.3 `SMA` vs. Other Lifecycle Models

This section compares the `SMA` to other well-known traditional and modern lifecycle models. Similarities, advantages, but also deficiencies of these models in comparison to the `SMA` are discussed, and we argue in which aspects the `SMA` is superior to the presented models and, to which application areas it is tailored.

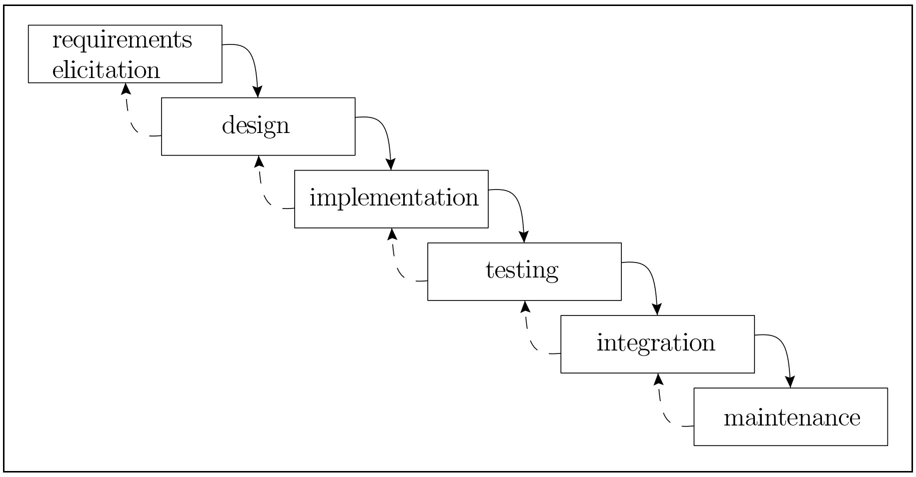


Figure 7.4: The Waterfall lifecycle model

7.3.1 The Waterfall- and V-model

Figure 7.4 gives a rough overview over the main phases that belong to the famous Waterfall model [Som06]. As mentioned before, major drawbacks of several traditional models like the Waterfall or V-model are:

(i) All requirements have to be fixed in advance.

(ii) As testing phases are scheduled at the end of the software development process, expensive and time-consuming backward steps are to be expected.

(iii) Typically, in industrial practice, during the development phase but especially during the maintenance phase, only the code is improved but the underlying documents and models are not extended or updated, to avoid overwhelming work. Then, however, significant problems arise if on basis of the current software a new version needs to be developed.

The `SMA`, however, overcomes the first problem by interactively deriving new scenarios while models evolve towards a final conforming and validated model. The second shortcoming is addressed by the intensive simulation and analysis phase on the design model level as well as due to application of model-based testing techniques to check conformance of the design model and the implementation model. The important fact is that any deficiency encountered can usually be formulated in terms of a (mis-)behavior, expressed as an MSC. In case the misbehavior is due to an invalid design model, it can be documented in the requirements documents as well as fed back to the learning phase to improve the design model. This on the one hand reduces the probability of having design or implementation flaws substantially, yet allows us to keep requirements and design models up-to-date. Similarly, during the maintenance phase, the modified behavior of the software system can be expressed by adding new MSCs or reclassifying previously classified MSCs to update the design model and corresponding design documents.

Note that `SMA` coincides with the Waterfall model and the V-model on major milestones of these lifecycle models, namely *requirements elicitation*, *design-model elaboration*, *implementation*, *testing* and *maintenance*.

As the next paragraphs will show, it also includes aspects of modern software engineering lifecycle models.

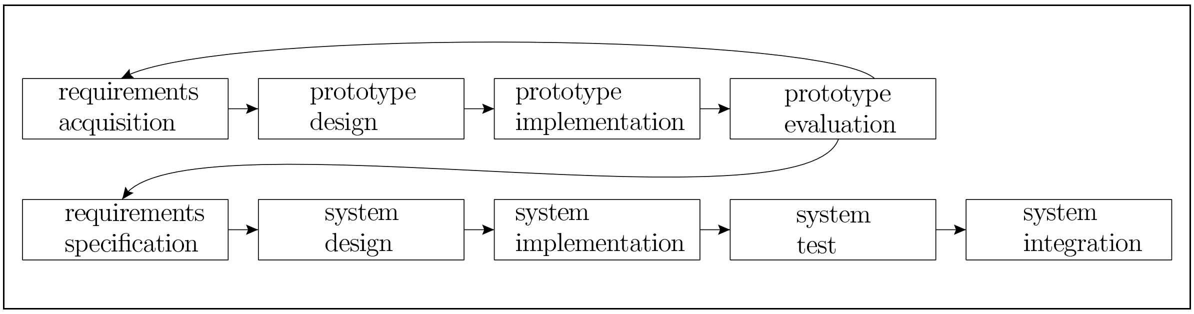


Figure 7.5: An evolutionary rapid prototyping lifecycle model

7.3.2 The Spiral Model

One of the first models which overcame the severe problems mentioned above was Boehm's *spiral model* [Boe88, Eas04, Som06]. For large software systems it is usually impossible to fix all requirements in advance. The spiral model therefore supports an iterative development of requirements and system prototypes, employing the following main phases: starting with the detection of goals, alternatives, and constraints, an evaluation of the alternatives and risks is performed until reaching a development and testing phase. Each cycle is concluded by the planning of the next iteration. It also allows for development of incremental versions of software resolving the third drawback of the Waterfall model.

The `SMA` adapted the *progressive* character of the spiral model but, to our opinion, has the extra benefit of easing the requirements elicitation and derivation of a design model: only a classification has to be provided. This significantly lowers the engineer's burden to define requirements. Nevertheless, the spiral model aims at developing large-scale projects while the main application area for the `SMA` is to be seen in developing software for embedded systems where the number of communicating entities is fixed a priori.

To benefit from both models one could also imagine to integrate the `SMA` partially into the spiral model. Parts of the system are then learned employing the `SMA` and the resulting components can afterwards be integrated in the overall system using the spiral model. In other words, the `SMA` would correspond to one iteration within the spiral model.

7.3.3 Rapid Prototyping

Rapid prototyping (RP) [Eas04, Som06] also resolves the traditional models' deficits of defining all requirements of a system in advance. This kind of software engineering lifecycle is employed for getting deeper insight on the requirements. In several iterations prototypes are generated (cf. Figure 7.5). The knowledge about requirements and design that is gained throughout these iterations is used as input for improving the prototypes of the next iteration. If a satisfactory prototype was created, it may serve as system specification, and the software development is continued by an iterative lifecycle model (e.g., using the Waterfall model) to build the final system. Note that analysis and testing phases can early be integrated into the process. This results in early feedback and, hence, less problems in later phases that would cause expensive redesign.

In software engineering several kinds of prototyping such as *throw-away-prototyping* and *evolutionary prototyping* are distinguished. In throw-away-prototyping a prototype is created, discussed with the client and, after an agreement about the correctness of the prototype, discarded again. The knowledge that was gained throughout the creation of

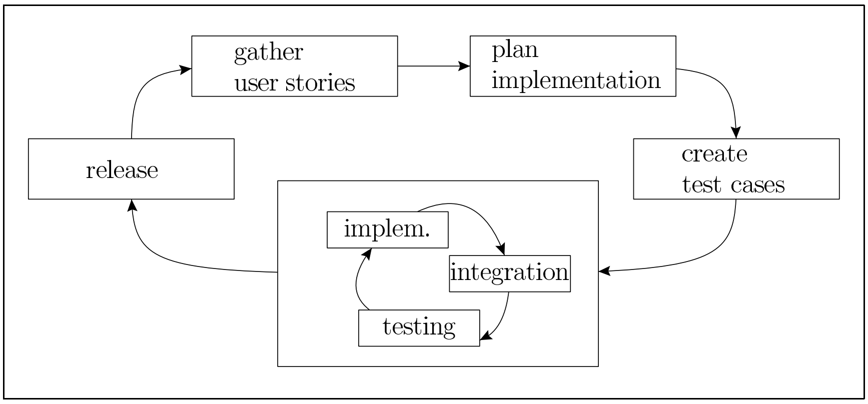


Figure 7.6: Extreme programming lifecycle model

the prototype and the discussion with the client are exploited to start the development of the final system with a different software engineering lifecycle.

Our lifecycle model very much resembles the *evolutionary prototyping* approach where a prototype is refined in several iterations, and in each iteration the knowledge acquired during the previous iterations is used to enhance the current prototype. This can also be regarded as a kind of risk-reduction technique where the probability of finding errors is increased by continuously learning about the system's requirements. However, as learning is based on exemplifying behavior that is expected to be documented within the learning process, the `SMA` does not suffer from the disadvantage of not having a formal requirements document as, due to time and financial constraints, it is usually the case in RP.

Moreover, bridging the gap from requirements to a design model—that apparently exists in rapid prototyping as well as in most software engineering lifecycle models—is a highly creative task that involves requirements as well as design engineers with expensive expertise. The `SMA`, however, is not necessarily dependent on highly experienced design personnel but rather on requirements engineers with domain knowledge because, as mentioned before, design questions are to a great extent solved automated by the learning algorithm, once architectural decisions have been fixed.

7.3.4 Agile Models

Agile models [AJ02] are a popular approach to iterative software development. They are mainly characterized by two properties: firstly, while the main focus in most traditional software development models is on creating documents at the end of each phase, in agile modeling this focus is changed towards immediate interaction of human beings. It is assumed that relying on formal documents only, is inappropriate and their production costly. Thus, a major goal of agile modeling is to replace documents by direct interaction of designer and client. Of course the one-sided point-of-view that generating documentation is expensive is not fully correct as formal documentation conserves knowledge gained in the current phase and thus eases the software development in later phases.

The `SMA` exhibits both, direct communication with the customer while inferring the design model, and automatic creation of formal documentation, i.e., the iterative update and completion of the requirements document.

The second fundamental characteristic of agile models is that, due to short development cycles, the model stays highly adaptive to changes in requirements.

These properties stress the flexibility of agile models compared to traditional software development models.

An example agile model that resembles the `SMA` to a certain extent is the so-called *extreme programming model* [AJ02]. As Figure 7.6 depicts, extreme programming uses collections of so-called *user stories* (i.e., scenarios). In each iteration some of these user stories are planned for implementation and, as in the `SMA`, an early integration of testing is provided, to obtain an early defect detection. After the test case creation phase a loop of implementing, integrating and testing is performed until converging to a release. This, in turn, can be analyzed again to gather new user stories closing the cycle to the next iteration. Similar to the `SMA`, regular and early testing phases are stipulated. Design and implementation flaws are usually detected early, allowing for substantial cost reduction and considerably shorter time-to-market phases. The designing and programming-in-pairs paradigm supported by both approaches is less error-prone and, thus, serves as a further risk-reduction technique, which lowers costs of possible redesign or reimplementation.

7.4 Exemplifying the `SMA`

In this section, we apply the `SMA` on a concrete yet simple example. More specifically, our goal is to derive a model for the well-known *alternating bit protocol* (ABP). Along the lines of [Lyn97, Tan02], we start with a short requirements description in natural language. Examining this description, we will identify the participating processes and formulate some initial MSCs exemplifying the behavior of the protocol. These MSCs will be used as input for `Smyle` which in turn will ask us to classify further MSCs, before deriving a first model of the protocol. After a second iteration, a further hypothesis is derived, and extensively simulated and tested. Finally, we come up with a design model for the ABP matching the model from [Tan02].

Problem Description

The main aim of the ABP is to assure the reliability of data transmission initiated by a *producer* through an *unreliable* FIFO (first-in-first-out) channel to a *consumer*. Here, unreliable means that data can be corrupted during transmission. We suppose, however, that the consumer is capable of detecting such corrupted messages. Additionally, there is a channel from the consumer to the producer, which, however, is assumed to be reliable. The protocol now works as follows: initially a bit `b` is set to 0. The *producer* keeps sending the value of `b` until it receives an acknowledgment message a from the consumer. This affirmation message is sent some time after a message of the producer containing the message content `b` is obtained. After receiving such an acknowledgment, the producer locally inverts the value of `b` and starts sending the new value until the next affirmation message is received at the producer. The communication can terminate after any acknowledgment a that was received at the producer side.

Applying the `SMA`

According to the `SMA`, we first start with identifying the participating processes in this protocol: the *producer* p and the *consumer* c.

Next, we turn towards the *scenario extraction phase* and have to come up with a set of initial scenarios. Following the problem description, we first derive the MSC shown in

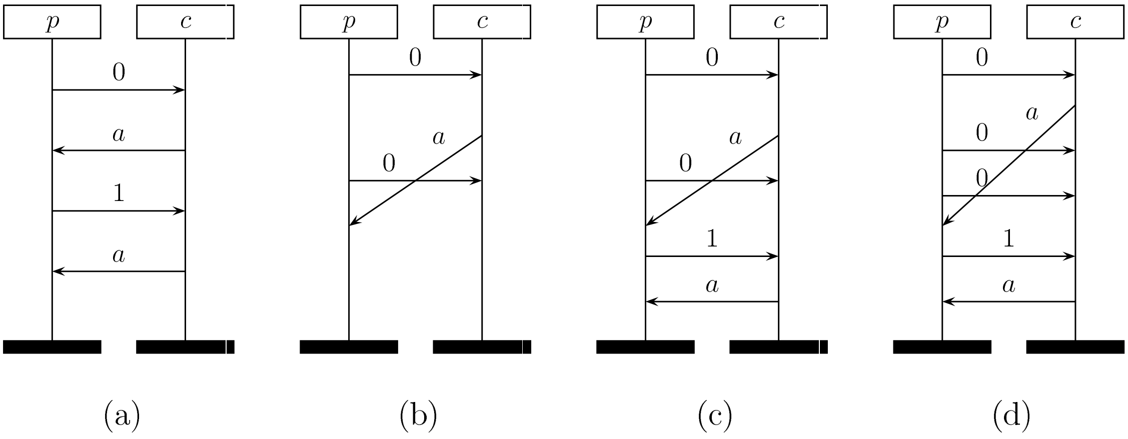


Figure 7.7: Four positive initial input scenarios for `Smyle`

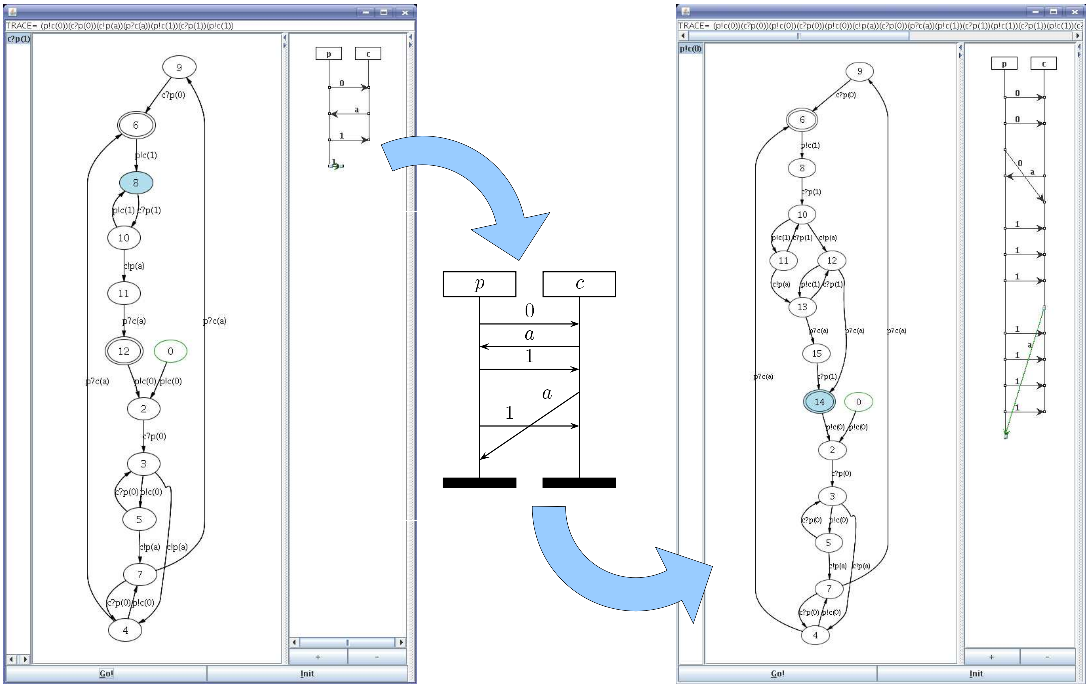
Figure 7.7(a). It describes that indeed p sends first 0, gets an acknowledgment from c, then sends 1, and finally gets a further acknowledgment. Let us now consider the behavior caused by the non-reliability of the channel. We could imagine that p sends a message 0 but, due to channel latency, does not receive a confirmation within a certain time bound and thus sends a second 0 while the first one is already being acknowledged by c. This yields the MSC in Figure 7.7(b).

The problem description tells us: *after each acknowledgment the bit* `b` *is inverted*. Thus, the previous scenario is extended by a second phase where `b` $= 1$ is sent and directly acknowledged, shown in Figure 7.7(c). To exemplify that, on the producer's side, more than one message can be corrupted, we derive a scenario that amplifies the previous one: We add one more message with content `b` $= 0$ to the first phase of scenario (c) yielding the scenario from Figure 7.7(d).

We start the learning phase and feed the charts to `Smyle`, proceeding to the second step in the design phase. Within this learning phase, `Smyle` asks us to classify further 44 scenarios—most of which we are easily able to negate—before providing a first hypothesis of the design model.

Now the simulation phase is activated (cf. Figure 7.8(a)), where we can test the current model. We execute several events as shown in the right part of Figure 7.8(a) and review the model's behavior. We come across an execution where, after an initial phase of sending a 0 and receiving the corresponding affirmation, we expect to observe a similar behavior as in Figure 7.7(b) (but now containing the message content `b` $= 1$). According to the problem description this is a feasible protocol execution but is not contained in our system, yet. Thus, we encountered a missing scenario. Therefore, instead of proceeding to the *synthesis and analysis* phase, we enter the *scenario extraction phase* again, formulate the missing scenario (cf. Figure 7.8(b)), and input it into `Smyle` as a counterexample to the current model.

As before, `Smyle` presents further MSCs that we have to classify: Among others, we are confronted with MSCs that (1) do not end with an acknowledgment (cf. Figure 7.9(a)) and with MSCs that (2) have two subsequent acknowledgment events (cf. Figure 7.9(c)). Both kinds of behavior are not allowed according to the problem description. We identify a pattern in each of these MSCs, by marking the parts of the MSCs as shown in Figure 7.9(a) and (c), yielding internally PDL formulae representing these patterns:



```
TRACE= (p!c(0))(c?p(0))(c!p(a))(p?c(a))(p!c(1))(c?p(1))(p!c(1))
```

```
TRACE= (p!c(0))(c?p(0))(p!c(0))(c?p(0))(p!c(0))(c!p(a))(c?p(0))(p?c(a))(p!c(1))(c?p(1))(p!c(1))(c?p(1))(p!c(1))(c?
```

(a) (b) (c)

Figure 7.8: `Smyle`’s simulation window:
(a) intermediate internal model with missing behavior
(b) missing scenario
(c) final internal model

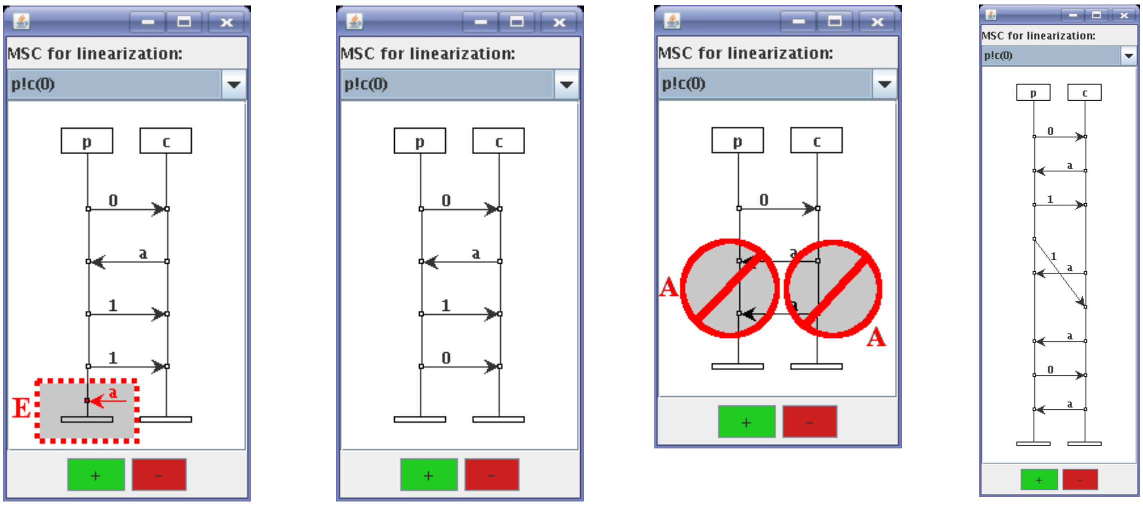
$$(1)\quad \mathbf{E}\big(\mathbf{procmax} \wedge ?(p,c,a)\big),$$

$$(2)\quad \mathbf{A}\big(\sqsubset\ ?(p,c,a);\ \mathsf{proc};\ ?(p,c,a)\ +\ !(c,p,a);\ \mathsf{proc};\ !(c,p,a)\ \sqsupset\ \mathbf{false}\big).$$

Note that in formula (2) the curly braces around actions have been omitted, again.

Instead of visually annotating MSCs, the formulae can also be directly entered via a dedicated formula editor. To tell `Smyle` to abolish all MSCs fulfilling the patterns we mark them as unwanted behavior. Thus, the MSCs from Figure 7.9(b) and (d) are automatically classified as negative later on. In addition, we reflect these patterns in the requirements documents by adding, for example, the explanation that *every message exchange has to end with an acknowledgment* and its formal specification (1). With the help of these two patterns, we continue our learning effort and end with the next hypothesis after a total of 55 user queries. Note that without adding these patterns, we would have needed 70 user queries. Moreover, identifying three more obvious patterns at the beginning of the learning process, we could have managed to infer the correct design model with only 12 user queries in total. Of course, one can argue that this is a high number of scenarios to classify, but this is the price one has to pay for getting an exact system and not an approximation (that indeed can be arbitrarily inaccurate) as in related approaches, e.g., [DLD05].

At the end of the second iteration through the learning phase, we are presented the simulation frame (Figure 7.8(c)) again. An intensive simulation does not give any evidence of wrong behavior. Thus, we enter the analysis phase to check the model with respect to further properties. For example, we check whether the resulting system can be implemented fixing a maximum channel capacity in advance. `MSCan` tells us that the system does not fulfill this property. Therefore, we need to add a (fair) scheduler to make the


MSC for linearization:
p!c(0)
p
c
0
a
1
1
E
a
+
-
MSC for linearization:
p!c(0)
p
c
0
a
1
0
+
-
MSC for linearization:
p!c(0)
p
c
0
A
a
A
+
-
MSC for linearization:
p!c(0)
p
c
0
a
1
1
a
a
0
a
+
-

(a) Always finally a receive of a on p

(b) Rejected by pattern from 7.9(a)

(c) Never two subsequent a on p or c

(d) Rejected by pattern from 7.9(c)

Figure 7.9: Patterns for (un)desired behavior

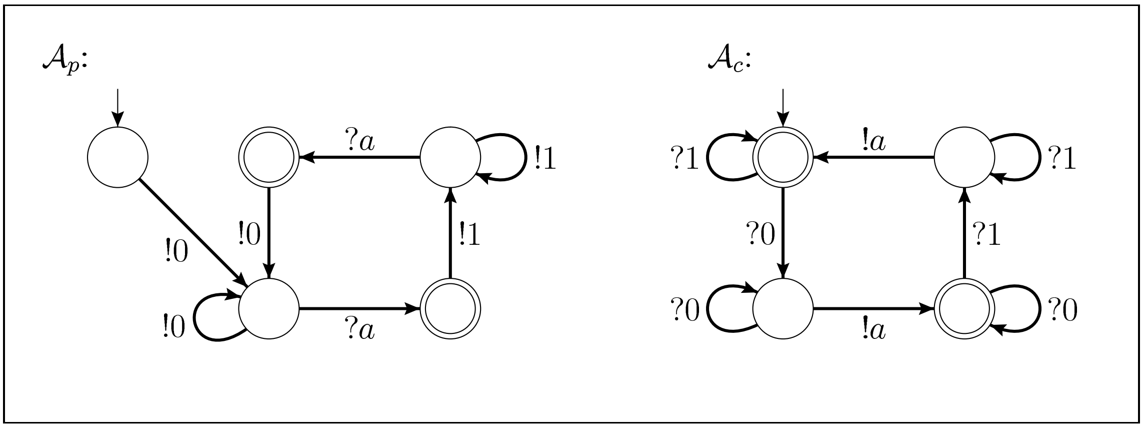

$\mathcal{A}_p$:
?a
!1
!0
!1
!0
!0
?a
$\mathcal{A}_c$:
?1
!a
?1
?0
?1
?0
?0
!a

Figure 7.10: CFM for the alternating bit protocol

protocol work in practice. According to the results from Chapter 6, a CFM is constructed which exactly is the one from Figure 7.10.

Part III

Tools and Applications

8 Learning Libraries

As indicated in Chapters 3 and 4, there is an uncountable number of learning algorithms available on the Internet. Of course the prementioned chapters do not give a complete overview over the field of off- and online learning algorithms, and there are many other important inference algorithms that had to be omitted. Usually, the most important and influential ones are implemented again and again by different researchers. Many of these learning algorithm implementations are *quick-and-dirty* applications and only meant to be a *proof-of-concept* for the authors' theoretical work. Moreover, most of these implementations are not available as source code to the general public. Thus, there is no possibility of extending or improving existing versions, or building new algorithms on basis of them, or to compare different versions thereof. The only available library heading for such aims was the `LearnLib C++` library [RS06, MRSL07] developed at the University of Dortmund. It implements Angluin's L^* algorithm, and some extensions thereof, and is capable of inferring DFA and Mealy machines, which are, roughly speaking, finite-state machines with output dependent on the current state. So-called *filters*, provided by `LearnLib` allow for exploiting domain-specific knowledge and, thereby, reduce the number of membership queries necessary to infer the target model. The `LearnLib` library contains some hard-coded filters, namely *prefix-closure*, *action-independence*, and *symmetry*, and allows for statistical methods to evaluate concrete learning instances in terms of number of states (of the minimal DFA), time-, and memory consumption. The `LearnLib` is available via an Internet connection to a server located at the University of Dortmund. In order to learn a model, a client has to connect to the password protected server, receives queries via the Internet, has to answer them locally, and sends them back to the server which subsequently evaluates them. This Internet-based information exchange persists until a hypothesis is returned by the server.

Initially, we integrated this library into our synthesis tool `Smyle`, which will be presented in Chapter 9. We were able to infer some small-size models using the `LearnLib`. Unfortunately, the service was unavailable several times due to a variety of reasons: some of them were: server problems at the `LearnLib` site, missing or unreliable Internet connections at conferences or universities, etc., or blocked ports which, because of the large number of security risks, is not unusual nowadays. Due to these reasons and the fact that the library is not available as source code or offline binary, for our purposes the `LearnLib` library could not be the ultimate choice. In search of alternatives we did not succeed and decided to launch a new project for an open-source learning library in the beginning of 2008: the `libalf` library [`libalf`]. In fact, at the beginning of 2009, we became aware of another learning library in the PhD thesis [Sha08]. It is called `RALT` which stands for *Rich Automata Learning and Testing*. In contrast to `LearnLib` and `libalf`, `RALT` was developed in `Java`. From the learning point of view it implements four learning algorithms, namely L^* and three slight variants [Nei03, BGJ$^+$05, Sha08] thereof for inferring Mealy machines. It employs minimal DFA and Mealy machines as hypotheses automata. Nevertheless, it neither includes extensions for domain specific requirements or to exploit domain specific features, nor is any information about external applications or statistics using `RALT` available in the literature.

Though our library and the `RALT` library partially overlap, the approaches are aiming at different goals. In contrast to `RALT`, the `libalf` library is meant to provide a learning suite with inference algorithms of all kinds (passive, active, offline, and online) such that end users can try to embed different algorithms provided by `libalf` into their application, compare the results, and finally employ the learning approach that is most suitable for their purposes.

In the following, we will describe `libalf`'s current development state, and give ideas how to extend it in the future.

8.1 `libalf`: The Automata Learning Framework

Our intention for launching the `libalf` project is to start a unifying learning framework with a clear and clean architecture and design for collecting learning algorithms of all kinds, providing the library's users with the `C++` source code allowing for an easy integration into their own projects, to extend existing algorithms, or to augment it by adding new algorithms. Moreover, we implemented the possibility of learning models via a dedicated server, and provide several interfaces to this client/server approach. In contrast to the `LearnLib`, we offer the possibility to install the `libalf` server on local machines, but of course also grant the possibility to connect to a central `libalf` server in analogy to the `LearnLib` idea. Initially, this project was initiated by the *Lehrstuhl für Informatik 2* (RWTH Aachen University), the *Laboratoire Spécification et Vérification* (ENS Cachan) and the *Institut für Informatik, Lehrstuhl IV* (TU Munich). As this project is meant to be open source, meanwhile other chairs and universities—like the *Oxford University Computing Laboratory* under supervision of Professor M. Kwiatkowska and the *Lehrstuhl für Informatik 7, RWTH Aachen University* under direction of Professor W. Thomas—evinced interest in participating in our learning framework and contributing new classes of learning algorithms. Moreover, there are already concrete plans for employing `libalf` for searching through source code for similar code fragments (so-called *clones*) by means of examples (E. Jürgens, TU Munich), or learning from log files of an existing system to derive a model for the underlying system (i.e., learning blackbox systems). The former idea has already been realized using the RPNI learning algorithm but, as results turned out to be unsatisfactory employing RPNI, the authors were looking for alternative learning approaches. Our library might be of help because it implements a variety of learning algorithms which may be more suitable.

8.2 Implementation Details

The `libalf` library is written in `C++` and currently contains approximately 12,500 SLoC[1]. It is available as source code from `http://libalf.informatik.rwth-aachen.de/`. Its extremely modular structure allows for easy extension in various ways. Our main objective is to provide means and pave the way for extending existing learning algorithms included in `libalf`, adding new algorithms on basis of already existing data structures and algorithms (new data structures can be integrated as well), and include user-defined filters and normalizers for exploiting domain-specific knowledge for reducing the number of queries in order to ease the learning task. An overview over the architecture of our

---

[1] SLoC (source lines of code) calcultaed using `sloccount`

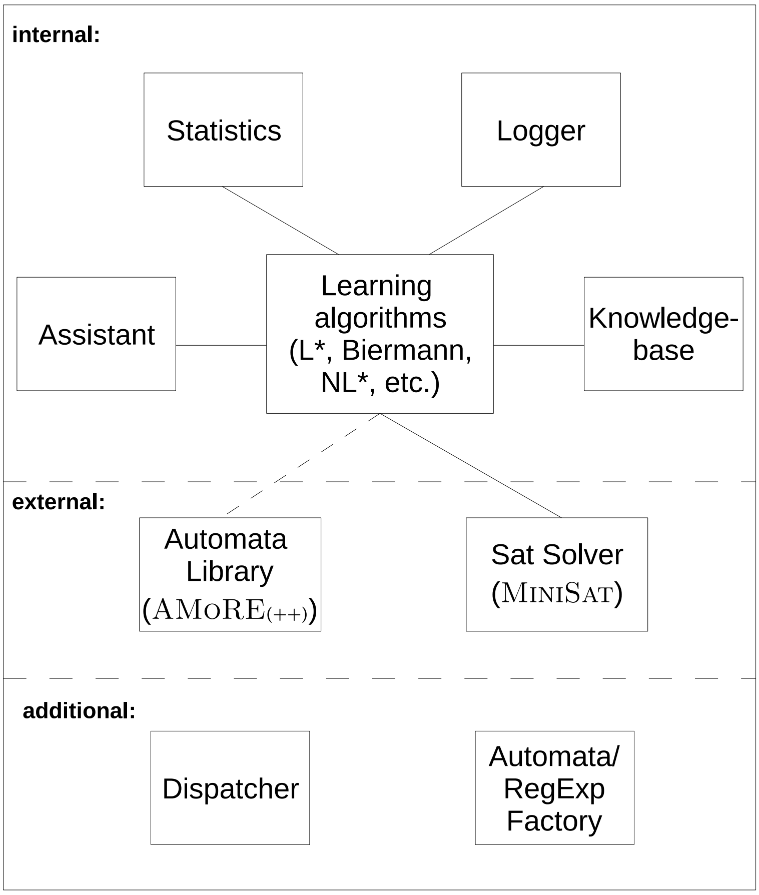

internal:
Statistics
Logger
Assistant
Learning algorithms (L*, Biermann, NL*, etc.)
Knowledge-base
external:
Automata Library (AMoRE(++))
Sat Solver (MiniSat)
additional:
Dispatcher
Automata/ RegExp Factory

Figure 8.1: Architecture of the `libalf` learning library

library is presented in Figure 8.1. It contains nine components (five internal, two external, and two additional components), which will subsequently be described in greater detail. Moreover, we implemented a `Java` interface for communicating with the *dispatcher* which is the server component of `libalf` providing means of remotely using `libalf` via a local network or the Internet, and a `Java` Native Interface (JNI)—by courtesy of Lehrstuhl für Informatik 7 (RWTH Aachen University)—for directly accessing the `C++` library from `Java` applications without using network communication via the dispatcher component. These interfaces, enriched by some small example applications, are also retrievable from the library's web page.

Learning algorithms Currently, the `libalf` library contains offline as well as active online learning algorithms. We implemented most learning algorithms presented in Chapters 3 and 4. Among them are the passive offline algorithms for deriving DFA, namely Biermann and RPNI and the active online algorithms L^* and L^*_{col} for inferring DFA, and NL^* for deriving canonical RFSA, respectively. The *learning algorithms* module collects the classes in which the learning algorithms are implemented. They have access to all other modules which are described below.

In the near future, we also plan to integrate the DeLeTe2 algorithm for offline learning (non-canonical) RFSA (cf. Subsection 3.2.1 on page 30) and a new online learning algorithm for inferring nondeterministic weak Büchi automata that is currently being developed. Moreover, as mentioned in the previous section, other parties agreed on contributing further learning techniques.

Knowledgebase The *knowledgebase* contains all information regarding the classification of examples asked and categorized so far, or initially provided by the user in case of passive offline algorithms. Moreover, the active online algorithms extend this container with questions that have to be answered by a teacher (e.g., a human user or the learning application itself). To this end, it maintains a list of unclassified words, which the teacher may iterate over, categorize them, and notify the learning algorithm about the successful classification. To prevent from redundantly storing data, the underlying data structure for already classified words is realized in an optimized tree-like fashion. Furthermore, we provide means for *undo* operations annihilating previous classifications. In some application domains (e.g., in the domain of our learning application `Smyle`, cf. Chapter 9) this feature might be a great advantage and substantially enhance usability.

Assistant The *assistant* module contains classes (namely filters and normalizers) that implement methods which can substantially reduce the number of queries posed by online algorithms by exploiting certain domain-specific knowledge. An example for such filters and normalizers is discussed below.

To clarify terminology: filters are methods that, given a word w and a set of classifications P, can determine whether there is a parity $p \in P$ such that the correct classification of word w is p. If the filter cannot decide that property for a word w, it is marked within the knowledgebase as “?” meaning that the answer cannot be derived autonomously, but has to be given by some teacher (e.g., a human user or a teacher implemented within the learning application). A classical example for a filter in the setting of MSCs would look as follows: being asked an MSC word w which is answered with parity p, the filter would store the corresponding MSC object $\mathscr{M}(w)$ and its classification p and, from that moment on, answer all equivalent MSC words $w' \in \mathit{Lin}(\mathscr{M}(w))$ according to answer p.

Another filter in this setting would directly reject all words that are not MSC words, and hence, for example, all words starting with a receive action, or ending with a send action.

In contrast to filters, normalizers are not only capable of autonomously answering certain queries but also of transforming the given word w into some normalized form $\eta(w)$ which is stored in the table. This way, the table only exhibits normalized words, thereby compressing the online algorithm's underlying table/data structure considerably. For a more detailed description of normalizers we refer to Chapter 5 and Section 6.3 where we introduced learning of congruence-closed languages and partial-order learning. As we will see later, the reductions concerning the memory consumption may be tremendous.

As stated in the beginning of this chapter, the `LearnLib` library also implements some filters. With this module we want to provide the means for writing user-defined assistants and share them with the learning community if the domain is of interest for a larger group of people. Currently, we have integrated a *normalizer* which is used for creating normal forms for MSC linearizations as described in Section 6.3. Statistical results for employing these normalizers can be found in Chapter 9.2.

Statistics The *statistics* module is meant to provide the `libalf` user with a wide variety of information about the learning task she has just performed. Currently, information about model size, number of (membership and equivalence) queries, memory consumption, time consumption, etc., are retrievable.

Logger The *logger* class provides means for obtaining additional (mostly debug) information about the current learning job. Given an a priori specified *level of detail*, the logger stores information about warnings and errors that occurred during the learning procedure. For the iterative **Biermann** learning approach (cf. Subsection 3.2.1 on page 25), for example, it logs which state sizes were already tested. Moreover, for all kinds of learning algorithms it detects if the user performs unexpected or illegal actions, whether or not an algorithm supports undo operations, in case of normal forms if a counterexample is evaluated to bottom, meaning that it is not part of the domain, or, for debugging purposes, a human readable version of the learning information an inference algorithm stores (e.g., the tables used in L^*, L^*_{col}, or NL^*).

Dispatcher The *dispatcher* is a server that provides the possibility of using the `libalf` facilities via a dedicated network protocol. This way, a user can connect to a (local or remote) machine on which the `libalf` server is installed and being executed, and perform the learning task in a client/server fashion. This is similar to the way the `LearnLib` library CORBA interface works. As, however, there is a known bug in the CORBA implementation of `Java`, it turned out that using the `LearnLib` CORBA interface is currently not suitable for learning larger size models. In contrast to the `LearnLib`, users can install the dispatcher on every computer they have access to, and are thus not dependent on a service being run on a remote server they cannot restart in case of hardware problems, etc.

In addition to an example which shows how to employ `libalf` in your own application using `C++` or a `Java` native interface, an example of how to use the dispatcher for a given learning task is given in the end of this section, too.

Automata and Regular Expression Factory The automata and regular expression factory, called `liblangen` (an abbreviation for *language generator*), is a stand-alone library for randomly generating automata and regular expression. It comprises a powerful library—developed in parallel to `libalf`—that is capable of automatically deriving large sets of automata in terms of DFA according to [CP05], NFA, and regular expressions as explained in [DLT04, BHKL08]. The user may specify certain criteria like number of models to infer, number of states of the equivalent minimal DFA, alphabet size, or distributions, e.g., over the connectives used in regular expressions, and eventually obtains a set of models according to these properties.

Additionally, we plan to maintain a database which is connected to this module, to store the derived sets of automata or regular expressions and the size in terms of the equivalent minimal DFA. The content of this database will be made accessible to public, and—once containing a reasonable number of examples—shall provide interested users with a great variety of automata sets tailored to their custom needs. Another advantage of such sets would be the ease of comparisons of existing learning algorithm implementations with new ones by executing them on the same input sample.

Automata Library Currently, some `libalf` test cases make use of the `AMoRE` and `AMoRE++` automata libraries. The former can be found at `http://amore.sourceforge.net/` whereas the latter is an extension of `AMoRE` which emerged in connection with the `libalf` project. `AMoRE` is an extensive library containing all standard automata operations (creating, extending, determinizing, minimizing, simulating automata (cf. `AMoRE++`), etc.). This library can be exchanged, augmented, or additional automata libraries can be easily integrated

```
Biermann testcase:
Logger log;
Knowledgebase knowledgeBase;
int  alphabetSize = 2;

// add sample sets to knowledgebase
knowledgeBase.addPositiveSamples(".0.1.",".0.0.0.",".1.1.1.",".1.1.1.0.1.");
knowledgeBase.addNegativeSamples(".0.",".0.0.",".1.",".1.1.");

MiniSat_biermann learner(knowledgeBase, log, alphabetSize);

DFA hypothesis;

if (!learner.advance(hypothesis))
  then
       print  ("advance returned false");
  else
       learner.print(cout);
```

Table 8.1: A pseudocode testcase for Biermann's algorithm

if new user requirements arise, e.g., for dealing with Büchi-, timed-, or probabilistic automata.

However, as the dashed line in Figure 8.1 signifies, there is currently no coupling between `libalf` and `AMoRE/AMoRE++` and only a loose coupling between the `libalf` test cases and `AMoRE/AMoRE++`. For the core functionality of `libalf`, however, no automata library is currently required.

Satisfiability Solver As described in paragraph 3.2.1, the Biermann offline learning algorithm can be implemented by transforming a constraint solving problem into a CNF formula which then can be checked for satisfiability using SAT solvers. As `MiniSAT` is an open-source award-winning satisfiability solver, we chose to integrate it into our tool for realizing the Biermann offline learning algorithm. In Table 8.1 a pseudocode implementation for using the Biermann algorithm (based on SAT solving) is depicted.

`MiniSAT` is freely available as source code or binary package from `http://minisat.se/`.

8.3 Learning Using the `libalf` Library

There are several possibilities how to use the `libalf` library in a user application. At the moment, we offer three alternatives: either the `C++` library is directly integrated into a `C++` application or the `Java` native interface is employed for embedding `libalf` into a `Java` application or a network connection to a server, on which `libalf` is installed, and the `libalf` server (i.e., the dispatcher) is running, is established, and an Internet-based learning session performed. In the following subsections we briefly describe the three alternatives. As our learning application `Smyle` currently makes use of the dispatcher, we will first elaborate on this component and afterwards briefly describe the other two approaches.

8.3.1 Learning Using the `libalf` Library's Dispatcher

The dispatcher is a service that provides the `libalf` functionality via a network connection. To this end, the `libalf` library and the dispatcher have to be installed on some local or remote machine. The dispatcher can be used in `C++` directly or via an additional `Java` interface which encapsules the network communication with the dispatcher. An example utilizing the additional `Java` interface is given in Table 8.2. We will briefly describe the main tasks that have to be performed to obtain a user-guided version of L*. First of all, the connection to the server has to be established by specifying a server address and a port (lines 5–9), and creating a `LearningTool` object. After instantiating an observation table (`obsTable`, line 12), which is an object gathering the functionality to communicate with the dispatcher, this observation table requests a session specifying the alphabet size, the learning algorithm (`Algorithm.ANGLUIN`), and a knowledgebase object (`knowledgebase`) (lines 13–17). Depending on the desired learning algorithm, either a set of input data is provided by the application and the server returns a hypothesis, which can be further processed by the application as in Table 8.1, or a continuous interaction of client and server is initiated, in which the application answers membership and equivalence queries posed by the instantiated learning algorithm running on the server as in Table 8.2. In the example of Table 8.2, we create an L* learning instance (`Algorithm.ANGLUIN`) and start the learning loop (line 19). To obtain a set of new queries, which must be user-classified, or to retrieve a hypothesis automaton, we call method `advance` on our observation table object (line 22). If this function does not return any conjecture (lines 25–36), we receive a set of queries (line 28) which have to be classified by the user (line 30) and uploaded to the dispatcher (line 31). In case method `advance` returned a conjecture, the user may either terminate the learning loop (lines 44, 45), or specify a counterexample (line 41) which then is sent to the dispatcher (lines 46, 47).

After at some point the user was satisfied with a hypothesis automaton, the learning loop is exited, and the learning session finally closed (line 52).

8.3.2 Learning Using the `libalf` Library Directly

In a `C++` application As the `libalf` library is developed in `C++`, integrating it into a `C++` application is the easiest of the three possibilities mentioned above. The only task that has to be performed is to link `libalf` against your own code. An example is given in Table 8.3.

Using JNI A further comfortable alternative, when developing a `Java` application that shall employ `libalf`, is to utilize the `Java` native interface (JNI). JNI is a standardized program interface to call platform specific methods. As such, `Java` applications using this interface are not platform independent anymore as long as the library they use is not available cross platform. Nevertheless, the communication with the native `C++` library is extremely fast yielding a convenient alternative to directly using `C++` code. We employ JNI to embed our learning library into `Java` applications. An example is given in Table 8.4.

To the best of our knowledge there are no comparable learning libraries comprising the learning algorithms our `libalf` library contains. Hence, it seems hard to come up with detailed empirical data. Note, however, that most of the statistics from Chapter 4 and the protocols from Chapter 9 were obtained employing the `libalf` library using the dispatcher component.

We hope that `libalf` will find broad acceptance among researchers, and that it will be the starting point for an extensive project in which different universities employ `libalf` or augment the library with new algorithms.

`libalf` Dispatcher example:

```
public static void main (String[] args)
{
    System.out.println(”Specify alphabet size: ”);
    int alphabetSize = readInt();
    String serverAddress = ”localhost” ;
    int serverPort = 23005 ;

    // Connect to the server
    LearningTool learningInstance = new LearningTool(serverAddress, serverPort) ;

    // Construct containers for observationTable, knowledgebase and hypotheses
    DFAObservationTable obsTable = learningInstance.createDFAObservationTable();
    Knowledgebase knowledgebase = new Knowledgebase();
    BasicAutomaton conjecture = null;

    // Initialize observationTable
    obsTable.init(alphabetSize, Algorithm.ANGLUIN, knowledgebase);

    while (true) // Start learning loop
    {
        // Next step in the learning algorithm
        conjecture = obsTable.advance();

        // Process membership queries if there was no conjecture, yet
        if (conjecture == null)
        {
            // Process unclassified elements in the knowledgebase
            for (int[] query : knowledgebase.getQueries())
            {
                boolean classification = getClassificationFromUser(query);
                knowledgebase.add_knowledge(query, classification);
            }

            // Return knowledgebase to dispatcher
            obsTable.processKnowledgebase(knowledgebase) ;
        }
        else // Process equivalence query in case of a conjecture
        {
            System.out.println(”Hypothesis automaton: ” + conjecture);

            int[] counterexample = getCounterexampleFromUser();

             // Is the conjecture is equivalent to the target regular language?
            if (counterexample == null)
                break; // Learning terminated successfully
            else // Return counterexample to dispatcher
                obsTable.addCounterexample(counterexample);
        }
    }
    // Destroy the observationTable and close the session
    obsTable.destroy() ;
}
```

Table 8.2: `Java` code for simulating L^* with human *Teacher* using the dispatcher (locally)

`libalf` `C++` EXAMPLE:

```
int main(int argc, char** argv)
{

    ostream_logger log(&cout, LOGGER_DEBUG);

    knowledgebase<bool> knowledge;

    int alphabet_size = 2;

    // create angluin_simple_table and let the user teach
    angluin_simple_table<bool> angluin(&knowledge, &log, alphabet_size);
    finite_automaton * conjecture = NULL;

    while (true)
    {
        // Next step in the learning algorithm
        conjecture = angluin.advance(f_is_dfa, f_alphabet_size, f_state_count,
                                     f_initial, f_final, f_transitions);

        // Process membership queries if there was no conjecture, yet
        if (conjecture == null)
        {
            // let user answer the queries from knowledgebase
            answer_knowledgebase(knowledgebase);
        }

        // Process equivalence query in case of a conjecture
        else
        {
            cout ≪ ”Hypothesis automaton: ” ≪ conjecture.generate_dotfile();

            list<int>* counterexample = getCounterexampleFromUser();

            // Is the conjecture equivalent to the target regular language?
            if (counterexample == NULL)
                break;
            else // return counterexample to the learning algorithm
                angluin.add_counterexample(counterexample);
        }
    }
}
```

Table 8.3: `C++` code for simulating L^* with human *Teacher* using `libalf` directly

`libalf` JNI EXAMPLE:

```
public static void main (String[] args)
{
    System.out.println(”Specify alphabet size: ”);
    int alphabetSize = readInt();

    // Construct a new knowledgebase and logger
    Knowledgebase knowledgebase = new Knowledgebase();
    BufferedLogger logger = new BufferedLogger();

    // Initialize learning algorithm
    LearningAlgorithm angluin = new AlgorithmAngluin(knowledgebase, alphabetSize, logger);

    // Initialize a hypothesis automaton
    BasicAutomaton conjecture = null;

    while(true) // Start learning loop
    {
        // Next step in the learning algorithm
        conjecture = angluin.advance();

        // Process membership query if there was no conjecture, yet
        if (conjecture == null)
        {
            // Process unclassified elements in the knowledgebase
            for (int[] query : knowledgebase.getQueries())
            {
                boolean classification = getClassificationFromUser(query);
                knowledgebase.add_knowledge(query, classification);
            }
        }

        // Process equivalence query in case of a conjecture
        else
        {
            System.out.println(”Hypothesis automaton: ” + conjecture);

            int[] counterexample = getCounterexampleFromUser();

            // Is the conjecture equivalent to the target regular language?
            if (counterexample == null)
                break; // Learning terminated successfully
            else // Return counterexample to learning algorithm
                algorithm.add_counterexample(counterexample);
        }

    }

}
```

Table 8.4: `Java` code for simulating L* with human *Teacher* using JNI

9 Synthesizing Models by Learning from Examples

In this chapter, we describe `Smyle` [`Smyle`], an application implementing major parts concerned with learning distributed systems, documented in Chapters 5 and 6 of this dissertation. We will describe `Smyle`'s usage from a user's perspective, and present some small to mid-size sample protocols we inferred using this application. Subsequently, we provide the reader with some information on the implementation, and close with further areas of application in which employing `Smyle` might be advantageous.

9.1 `Smyle` from a User Perspective

We have implemented the learning approach presented in Chapters 5, 6, Section 7.1, and Subsection 7.2.2 within our application `Smyle` which is freely available from our website located at `http://www.smyle-tool.org/`.

`Smyle` is an acronym for *Synthesizing Models bY Learning from Examples*. Its major objective is to ease the development of concurrent systems. More specifically, the overall goal is to derive communication models of concurrent systems in terms of CFMs (cf. Definition 6.1.2 on page 87).

Roughly speaking, the synthesis process starts by providing the tool with a set of sample MSCs for which each MSC has to be classified as either positive or negative. Positive MSCs describe system behavior that is *possible* whereas negative MSCs characterize *unwanted* or *forbidden* behavior. `Smyle` focuses on basic-MSC features like asynchronous message exchange, and forbids to deploy the complete MSC standard [ITU04]—which allows for alternation, loops, etc.—on purpose: the more expressive a specification language gets, the less intuitive and manageable it becomes. Simple pictures, however, are easy to understand and easy to supply. As mentioned in the rationale from Chapter 7, human beings prefer to describe system runs by means of simple examples and pictures. Basic MSCs just constitute such a device.

To sum up, we will have several types of queries within the `Smyle` learning approach that can either automatedly be answered by the tool or have to be resolved by the `Smyle` user.

(i) *Membership queries* are questions posed by the *Learner*. They are mostly answered automatically by our procedure.

(ii) Moreover, we have *user queries*. User queries (i.e., MSC pictures) have to be answered manually by the user (as long as we do not use PDL formulae to ease this task). To this end, the user has to classify the automatically presented MSCs as either positive or negative scenarios.

(iii) At last, the approach exhibits *equivalence queries*. Equivalence queries are rather rare and have to be answered by testing and simulating the hypothesis. For this pur-

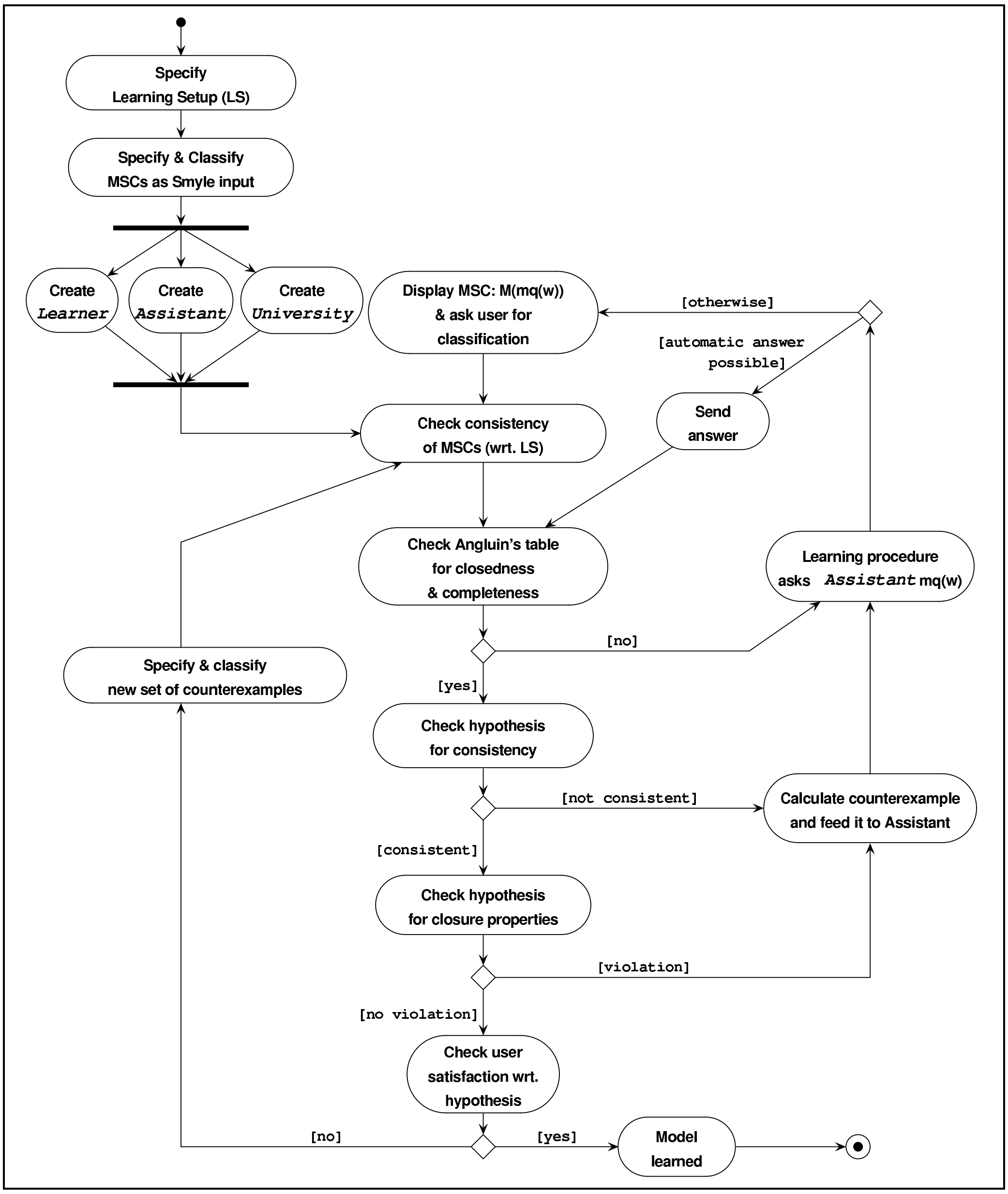

Specify
Learning Setup (LS)
Specify & Classify
MSCs as Smyle input
Create Learner
Create Assistant
Create University
Display MSC: M(mq(w))
& ask user for
classification
[otherwise]
[automatic answer
possible]
Send
answer
Check consistency
of MSCs (wrt. LS)
Check Angluin's table
for closedness
& completeness
Learning procedure
asks Assistant mq(w)
[no]
[yes]
Specify & classify
new set of counterexamples
Check hypothesis
for consistency
[not consistent]
Calculate counterexample
and feed it to Assistant
[consistent]
Check hypothesis
for closure properties
[violation]
[no violation]
Check user
satisfaction wrt.
hypothesis
[no]
[yes]
Model
learned

Figure 9.1: Activity diagram for the *learning chain* `Smyle` is based upon

pose, `Smyle` features a dedicated simulation component. If the user is not satisfied with the hypothesis she has to provide at least one counterexample MSC.

In the following we will describe in detail how `Smyle` can be used to derive protocol implementations.

9.1.1 The Learning Chain:

In this paragraph the approach for learning protocols, called the *learning chain*, with `Smyle` is described in-depth. Moreover, in Figure 9.1 a UML activity diagram depicts this learning chain in detail.

Let us now focus on explicating this learning chain. In order to initiate the synthesis process, at first, the user is asked to specify the learning setup (cf. Definition 6.2.1 on page 95). Having selected a language/CFM type (universally bounded, existentially

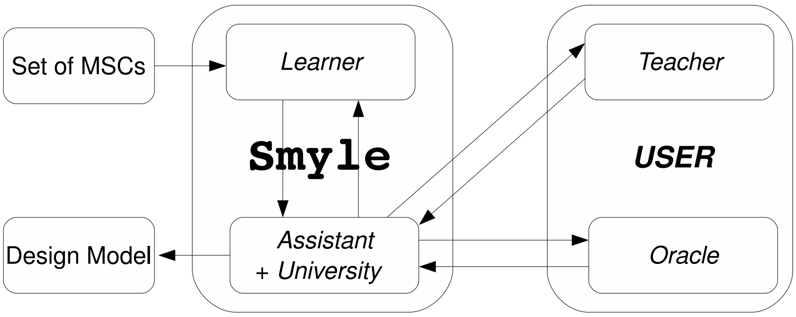
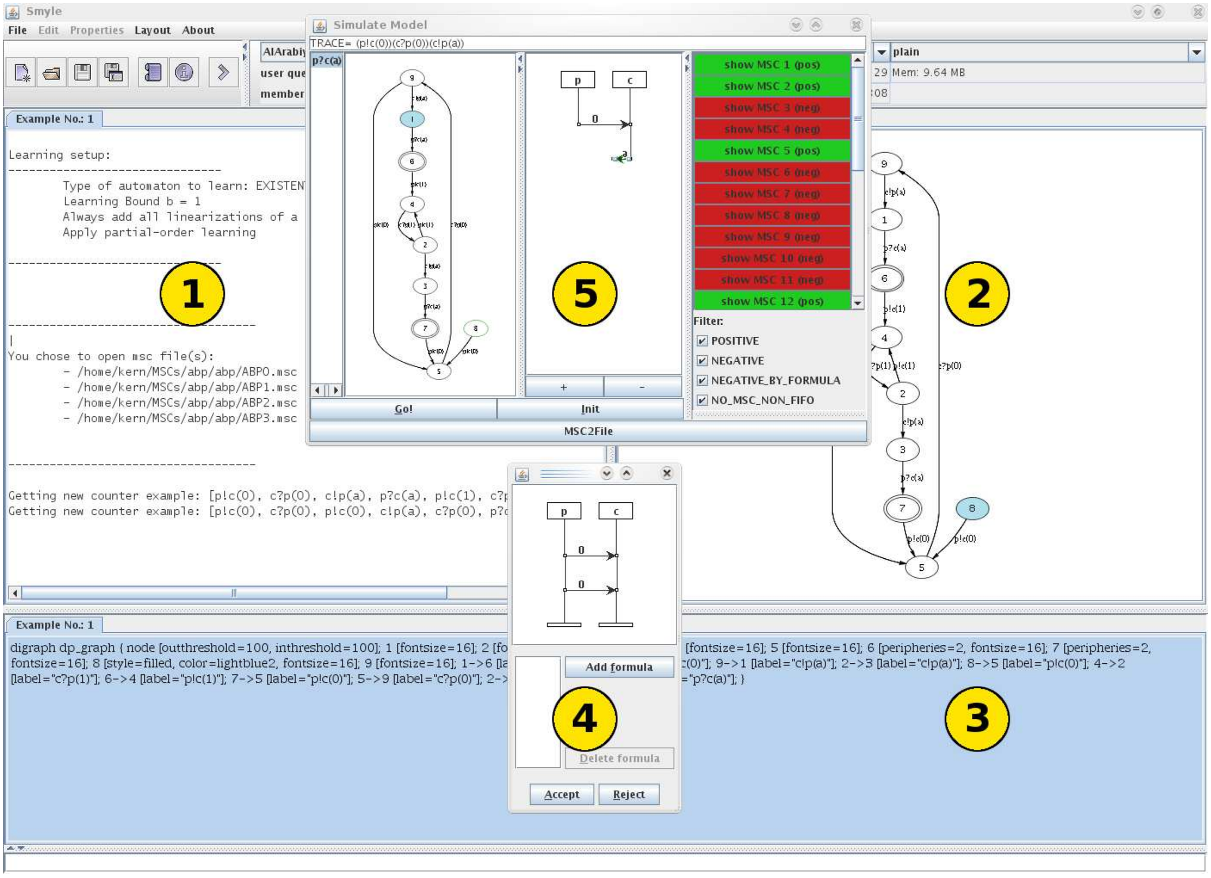


Figure 9.2: `Smyle`'s architecture: learning overview

B-bounded, or deterministic universally bounded deadlock-free weak), we can choose whether or not to apply partial-order learning (cf. Chapter 5 and Section 6.3), and have to provide a channel bound $B \in \mathbb{N}$ in case of an existentially-bounded learning setup. Thereafter, the user has to provide an initial set of MSCs. These MSC specifications must then be divided into *positive* (i.e., MSCs contained in the language to learn) and *negative* (i.e., MSCs not contained in the language to learn) samples. After submitting these examples, all MSCs are checked for consistency with respect to the properties of the learning setup and the *Learner*, *Assistant*, and *University* are created. The *Learner* executes the main learning loop in which the questions from the learning algorithm are being answered. The *Assistant* stores answers to already classified MSCs and verifies MSC properties like FIFO, B-boundedness etc. Thereby it acts as a filter for membership queries. Moreover, the *University* represents the interface between graphical user interface and learning components, and additionally stores information about the learning setup. MSCs violating the properties of the learning setup are stored as negative examples. Now the learning algorithm starts. As presented in the learning overview in Figure 9.2, the *Learner* continuously communicates with the *Assistant* and the *University* in order to gain answers to membership queries. This procedure halts as soon as a query cannot be answered by the *Assistant*. In this case, the *Assistant* forwards the inquiry to the user, displaying the MSC in question on the screen (cf. Figure 9.3 (4)). The user must then classify the MSC as either positive (Accept) or negative (Reject). The *Assistant* checks the classification for validity wrt. the learning setup. Depending on the outcome of this check, the current MSC is assigned to the positive or negative set of possible future queries (called *Pool* in Figure 6.6 on page 96 and Tables 6.1 and 6.2 on pages 98f.). Note that, as an MSC can have many linearizations that all might be asked during the learning procedure, we have to store the classified MSC in order to automatically categorize equivalent linearizations later on. Moreover, the user's answer is passed to the *Learner* which then continues its question-and-answer procedure with the *Assistant*. If the learning library `libalf` proposes a hypothesis automaton (cf. Figure 9.3 (2) and its corresponding `Dot` representation in component (3)), the *Assistant* tests whether the learned model is conform with all queries that have been categorized but not yet been asked. If a counterexample is encountered, it presents it to the learning algorithm which, in turn, continues the learning procedure until the next possible solution is found. Moreover, hypotheses are always checked for the closure properties, like domain closure, equivalence relation ($\approx_{\mathcal{D}}$) and the diamond property presented in the proof of Theorem 6.2.8 on page 101. As soon as any indication of a violating run occurs, the corresponding trace is returned to the learning procedure as counterexample. Contemporaneously, component (1) of Figure 9.3) logs these counterexamples. In case there is no further evidence for contradicting samples,


Smyle
File
Edit
Properties
Layout
About
Simulate Model
TRACE= (p!c(0))(c?p(0))(c!p(a))
p?c(a)
Example No.: 1
1
2
3
4
5
show MSC 1 (pos)
show MSC 2 (pos)
show MSC 3 (neg)
show MSC 4 (neg)
show MSC 5 (pos)
show MSC 6 (neg)
show MSC 7 (neg)
show MSC 8 (neg)
show MSC 9 (neg)
show MSC 10 (neg)
show MSC 11 (neg)
show MSC 12 (pos)
Filter:
☑ POSITIVE
☑ NEGATIVE
☑ NEGATIVE_BY_FORMULA
☑ NO_MSC_NON_FIFO
Go!
Init
MSC2File
Add formula
Delete formula
Accept
Reject
plain
Mem: 9.64 MB
p
c
0

```
Learning setup:
--------------------------------
        Type of automaton to learn: EXISTEN
        Learning Bound b = 1
        Always add all linearizations of a
        Apply partial-order learning

--------------------------------

-------------------------------------
|
You chose to open msc file(s):
       - /home/kern/MSCs/abp/abp/ABP0.msc
       - /home/kern/MSCs/abp/abp/ABP1.msc
       - /home/kern/MSCs/abp/abp/ABP2.msc
       - /home/kern/MSCs/abp/abp/ABP3.msc

-------------------------------------

Getting new counter example: [p!c(0), c?p(0), c!p(a), p?c(a), p!c(1), c?p
Getting new counter example: [p!c(0), c?p(0), p!c(0), c!p(a), c?p(0), p?c
```

```
digraph dp_graph { node [outthreshold=100, inthreshold=100]; 1 [fontsize=16]; 2 [fo
fontsize=16]; 8 [style=filled, color=lightblue2, fontsize=16]; 9 [fontsize=16]; 1->6 [la
[label="c?p(1)"]; 6->4 [label="p!c(1)"]; 7->5 [label="p!c(0)"]; 5->9 [label="c?p(0)"]; 2->
[fontsize=16]; 5 [fontsize=16]; 6 [peripheries=2, fontsize=16]; 7 [peripheries=2,
c(0)"]; 9->1 [label="c!p(a)"]; 2->3 [label="c!p(a)"]; 8->5 [label="p!c(0)"]; 4->2
="p?c(a)"]; }
```

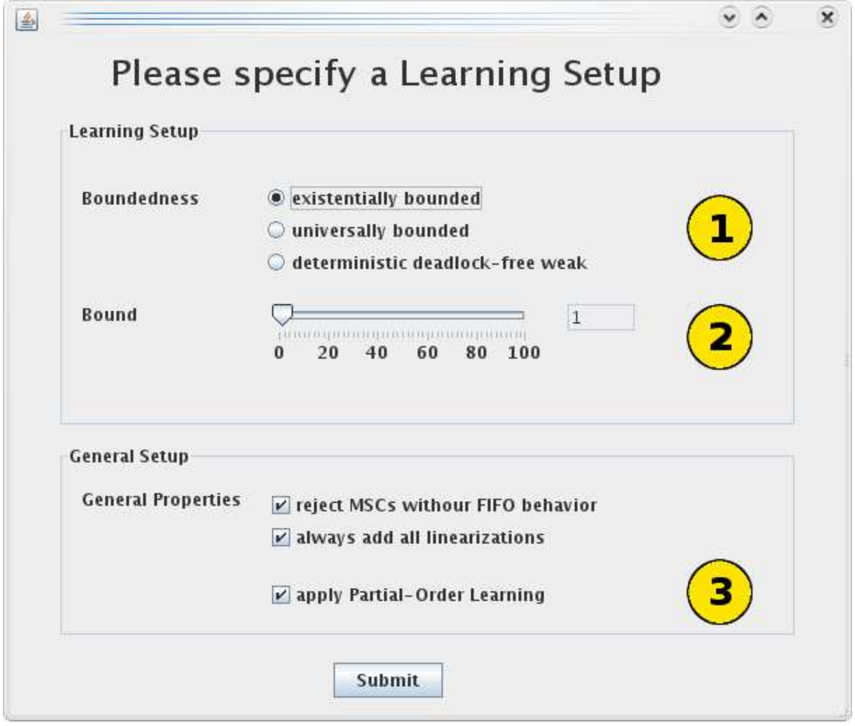
Figure 9.3: `Smyle`'s graphical user interface

a new frame appears (cf. Figure 9.3 (5)). Among others, it visualizes the currently learned automaton (2), as well as a panel for displaying MSCs of runs of the system described by the automaton. Panel (5) can be employed to simulate the behavior of the learned protocol. While executing actions and, thus, traversing the hypothesis automaton, component (5) constructs the underlying message sequence chart of the current run of the hypothesis automaton. This procedure can be repeated until the protocol designer (i.e., the `Smyle` user) comes to a conclusion about the correctness of the system. After getting enough impression on the protocol's behavior, the user closes the simulation window, and is then asked if she agrees with the solution. She may either stop, or introduce a new set of counterexamples proceeding with the learning procedure.

We will now elaborate on the main phases just introduced.

Specifying a Learning Setup

Specifying a learning setup mainly comprises three tasks. Firstly, the user has to specify which language type she intends to infer (cf. Figure 9.4 (1)). Within `Smyle` we have implemented three of the language types described in Chapter 6. The `Smyle` user may choose between *universally bounded*, *existentially B-bounded* or *deterministic universally bounded deadlock-free weak* output automata. For the second case a bound $B \in \mathbb{N}$ has to be user-specified (cf. Figure 9.4 (2)). In case the user opts for a universally B-bounded system, she will obtain an automaton that is realizable using finite channel capacity B resulting in a finite-state system. Selecting an existentially B-bounded learning setup



Please specify a Learning Setup

Learning Setup

Boundedness
● existentially bounded
○ universally bounded
○ deterministic deadlock-free weak

Bound: 1

General Setup

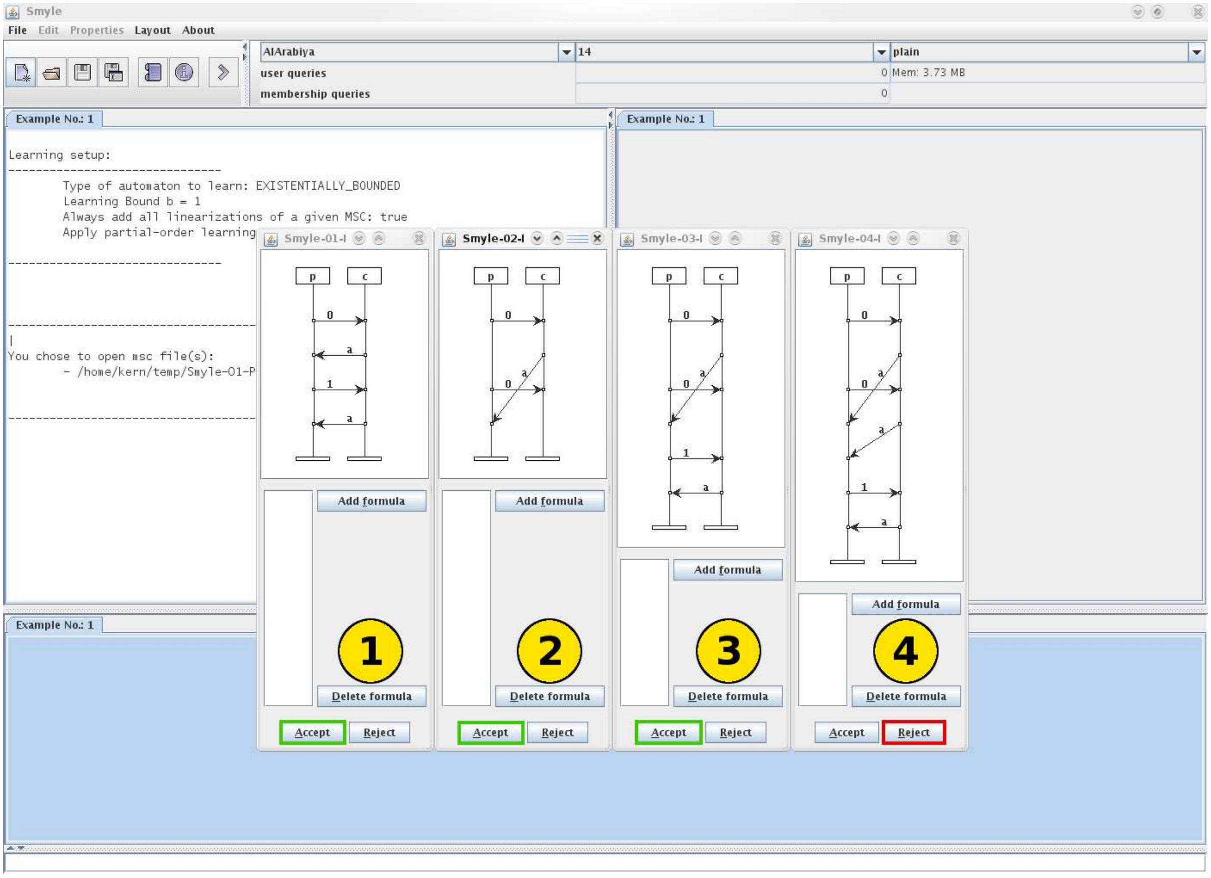
General Properties
☑ reject MSCs withour FIFO behavior
☑ always add all linearizations
☑ apply Partial-Order Learning

Figure 9.4: Choosing a learning setup in `Smyle`

is useful for inferring possibly infinite-state systems because only existential bounds are required for the system's channels. An existentially B-bounded learning setup allows the system developer to include system behavior that may exceed the system's channel sizes but, at the same time, guarantees that there is at least one execution (i.e., a total ordering of events viz. a linearization) of each classified MSC that adheres to this limit. Therefore, an appropriate scheduler will always be able to execute the *good* linearizations (i.e., runs not exceeding bound B) and disregard the ones going beyond this bound. The third category of automata derivable by `Smyle` are deterministic universally bounded deadlock-free weak CFMs. Choosing this type of learning setup will infer a CFM that recognizes a product MSC language as described in Chapter 6. A further adjustable parameter is to enable or disable the partial-order learning approach (cf. Figure 9.4 (3)). As described in Chapter 5 and Section 6.3, turning on partial-order learning may substantially reduce the amount of memory in use, as well as positively influence the number of membership queries posed. Therefore, this option is always enabled by default, but can be turned off on user demand. In Figure 9.4 `Smyle`'s learning setup frame is shown for a concrete learning setup. In this example we chose an existentially 1-bounded learning setup employing partial-order learning.

Having selected a learning setup, `Smyle` should be supplied with an initial set of MSCs. In the example (cf. Figure 9.5) we choose the three left-most MSCs (1)-(3) to be positive and the right-most scenario (4) to be negative. Subsequently, `Smyle` will ask the user to classify these MSCs and start the learning procedure thereafter (cf. Figure 9.2). This specification phase is supported by a dedicated MSC editor, which will now briefly be described.

The MSC editor: In connection with our learning application `Smyle`, we are also developing an MSC editor—as integrated `Smyle` component, and as stand-alone application—



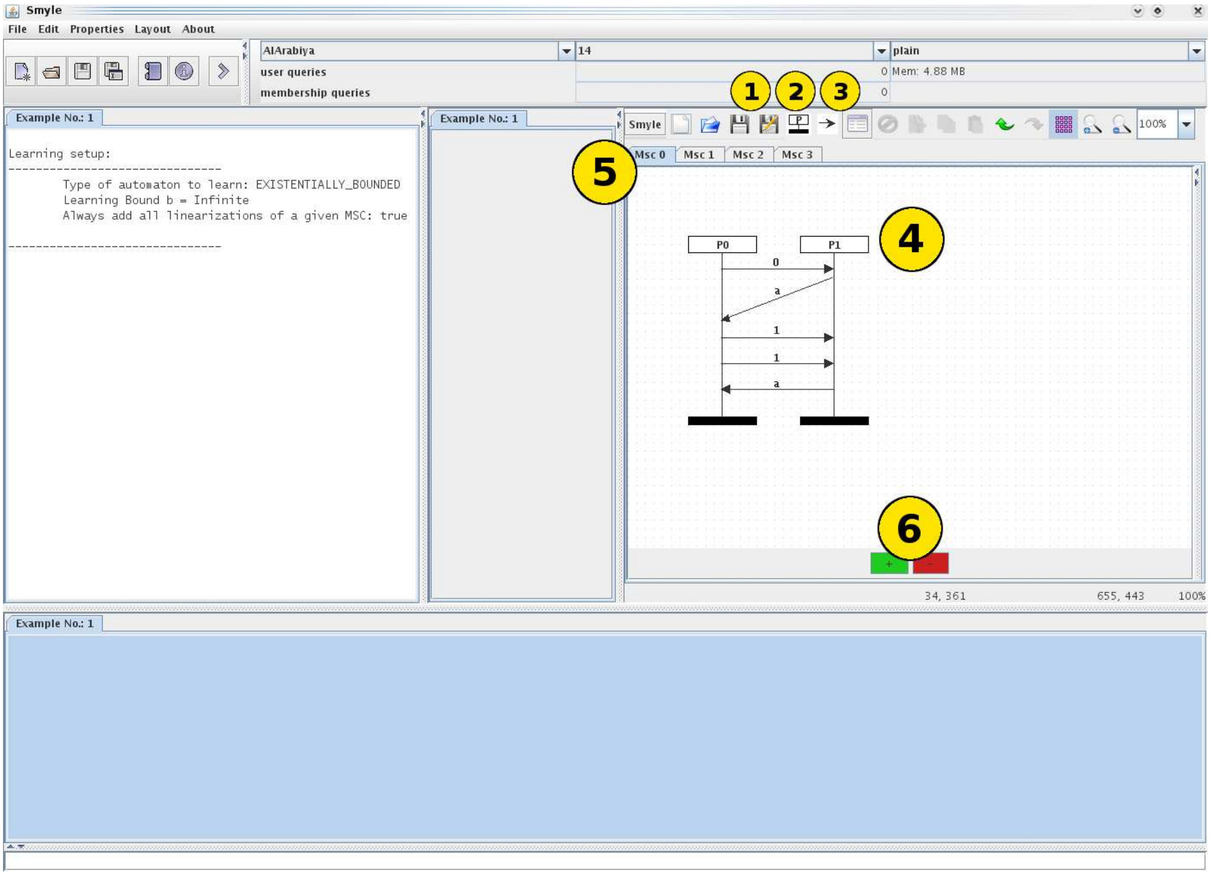
```
Learning setup:
--------------------------------
        Type of automaton to learn: EXISTENTIALLY_BOUNDED
        Learning Bound b = 1
        Always add all linearizations of a given MSC: true
        Apply partial-order learning

--------------------------------



--------------------------------------------
|
You chose to open msc file(s):
        - /home/kern/temp/Smyle-01-P


--------------------------------------------
```

Smyle
File
Edit
Properties
Layout
About
AlArabiya
14
plain
user queries
0
membership queries
0
Mem: 3.73 MB
Example No.: 1
Example No.: 1
Example No.: 1
Smyle-01-I
Smyle-02-I
Smyle-03-I
Smyle-04-I
p
c
0
a
1
Add formula
Delete formula
Accept
Reject
1
2
3
4

Figure 9.5: Choosing and classifying a set of input MSCs for `Smyle`

for graphically creating and modifying MSCs. Currently, the MSC editor is in a prototypical development state. Nevertheless, among others, it already features all means to comfortably specify basic MSCs, which serve as input for `Smyle`.

Whenever new MSCs have to be specified in order to start or continue the learning phase, `Smyle` can either load MSC documents containing basic MSCs from the file system, or offer to use the optionally integrated MSC editor (cf. Figure 9.6) for easy specification of basic MSCs. Let us briefly describe the main functionality of this editor component. It contains a menu panel for loading and saving MSCs from and to the file system (cf. Figure 9.6 (1)), buttons for creating new processes within an MSC (cf. item (2)), specifying messages (cf. item (3)), and other helpful features (e.g., undo and redo operations, zooming, etc.). Moreover, in the main panel (cf. item (4)) the desired MSC can be drawn and altered on demand. The user has the possibility to specify various scenarios using the tabbed pane (cf. item (5)). After finishing the specification task, the MSCs can be classified using the buttons at the bottom of this component (cf. item (6)) where the green button denotes positive behavior and the red button negative behavior. After classifying all MSCs, they can directly be fed back to `Smyle` in order to start or continue the learning chain. The editor also provides functionality for storing MSCs in many different formats (e.g., LaTeX, `fig`, etc.). An extended, stand-alone prototype version of the editor covering the ITU Z.120 standard to a large extend will also be available in near future at the `Smyle` homepage [`Smyle`].



user queries 0
membership queries 0

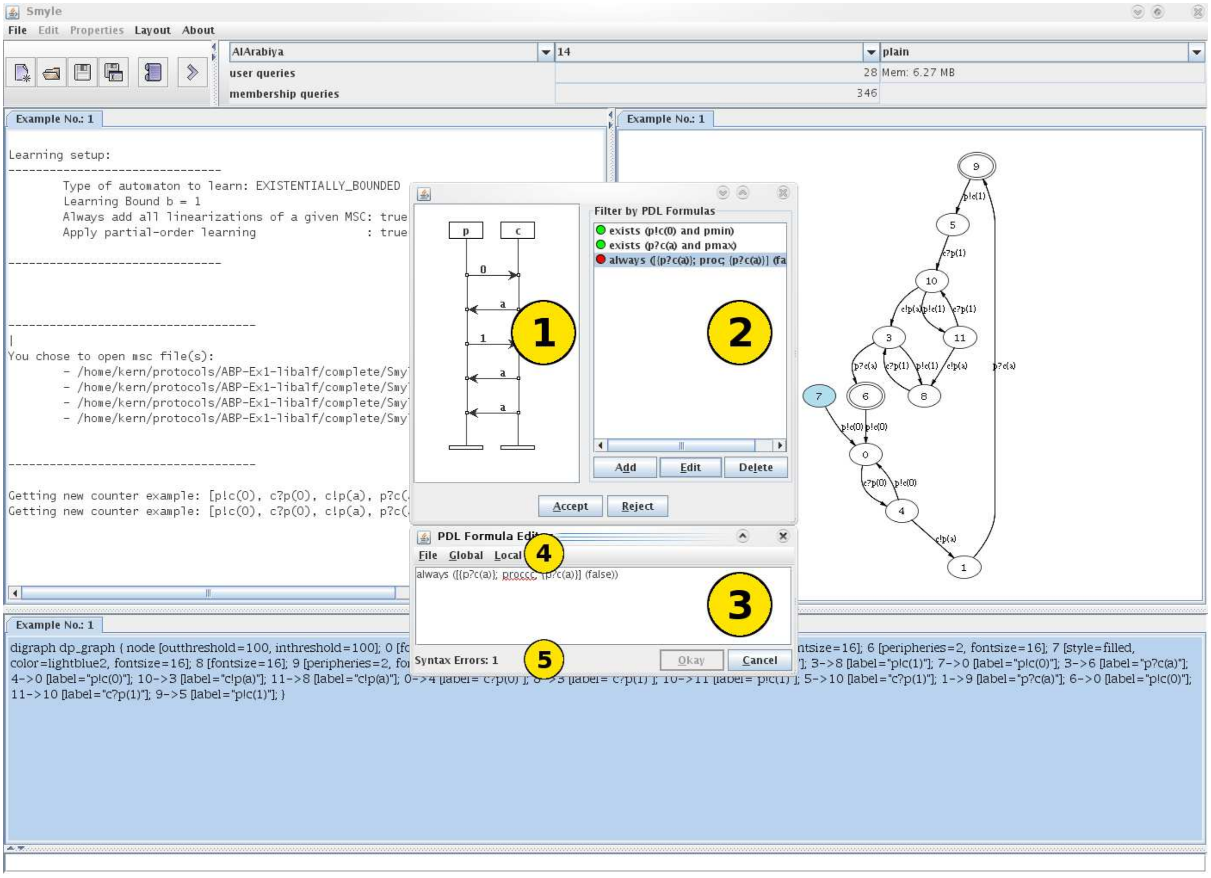
```
Learning setup:
-------------------------------
        Type of automaton to learn: EXISTENTIALLY_BOUNDED
        Learning Bound b = Infinite
        Always add all linearizations of a given MSC: true

-------------------------------
```

Figure 9.6: MSC editor integrated into Smyle

Performing Classifications

Successively, new MSCs as depicted in Figure 9.3 (4) or 9.5 (1)-(4) are presented to the user (in this phase acting as *Teacher*) who in turn has to classify these scenarios as either wanted (Accept) or illegal (Reject). As described in Subsection 7.2.2 in this phase patterns are usually detected and can be added to Smyle via the integrated formula editor (cf. Figure 9.7). This eases the user's categorization task considerably, as many classifications can then be performed autonomously by the tool.

Easing the learning process: In order to simplify the user's task of classifying scenarios, Smyle contains means for specifying PDL formulae over MSCs (cf. Section 7.1), a simple logic that—as we proved in Theorem 7.1.4 on page 115—comes with an efficiently solvable membership problem. Like MSCs, PDL formulae are used to express desired or illegal behavior, but in a broader sense. They are to be seen as general rules which apply for *all* runs of the system (and not only all executions of one scenario). Hence, if a user detects generally wanted or unwanted properties of the presented MSCs, she may specify formulae which express these *generics*. Smyle is supplied with these formulae and can, from that moment on, accept or reject all MSCs that fulfill or, respectively, violate one of these formulae. This technique reduces the number of user queries substantially. An example formula is $\varphi = \mathbf{A}(\sqsubset \{?(p,c,\mathtt{a})\};\ \mathsf{proc};\ \{?(p,c,\mathtt{a})\} \sqsupset\ \textbf{false})$ which states that there must not be two subsequent occurrences of the same receive action (i.e., $?(p,c,\mathtt{a})$) on the same process p. Hence, if formula φ is fed to Smyle as negative generic, all MSCs featuring this behavior, e.g., the one in Figure 9.7 (1), would be regarded as negative



AlArabiya | 14 | plain

user queries 28 Mem: 6.27 MB

membership queries 346

```
Learning setup:
--------------------------------
        Type of automaton to learn: EXISTENTIALLY_BOUNDED
        Learning Bound b = 1
        Always add all linearizations of a given MSC: true
        Apply partial-order learning                : true

--------------------------------



-------------------------------------
|
You chose to open msc file(s):
        - /home/kern/protocols/ABP-Ex1-libalf/complete/Smy
        - /home/kern/protocols/ABP-Ex1-libalf/complete/Smy
        - /home/kern/protocols/ABP-Ex1-libalf/complete/Smy
        - /home/kern/protocols/ABP-Ex1-libalf/complete/Smy


-------------------------------------

Getting new counter example: [p!c(0), c?p(0), c!p(a), p?c(
Getting new counter example: [p!c(0), c?p(0), c!p(a), p?c(
```

Filter by PDL Formulas

- ● exists (p!c(0) and pmin)
- ● exists (p?c(a) and pmax)
- ● always ([{p?c(a)}; proc; {p?c(a)}] (fa

```
always ([{p?c(a)}; proccc; {p?c(a)}] (false))
```

Syntax Errors: 1

```
digraph dp_graph { node [outthreshold=100, inthreshold=100]; 0 [fo                    ntsize=16]; 6 [peripheries=2, fontsize=16]; 7 [style=filled,
color=lightblue2, fontsize=16]; 8 [fontsize=16]; 9 [peripheries=2, fo                 ]; 3->8 [label="p!c(1)"]; 7->0 [label="p!c(0)"]; 3->6 [label="p?c(a)"];
4->0 [label="p!c(0)"]; 10->3 [label="c!p(a)"]; 11->8 [label="c!p(a)"]; 0->4 [label="c?p(0)"]; 8->3 [label="c?p(1)"]; 10->11 [label="p!c(1)"]; 5->10 [label="c?p(1)"]; 1->9 [label="p?c(a)"]; 6->0 [label="p!c(0)"];
11->10 [label="c?p(1)"]; 9->5 [label="p!c(1)"]; }
```

Figure 9.7: Specifying some sample PDL formulae in `Smyle`

samples without questioning the user anymore.

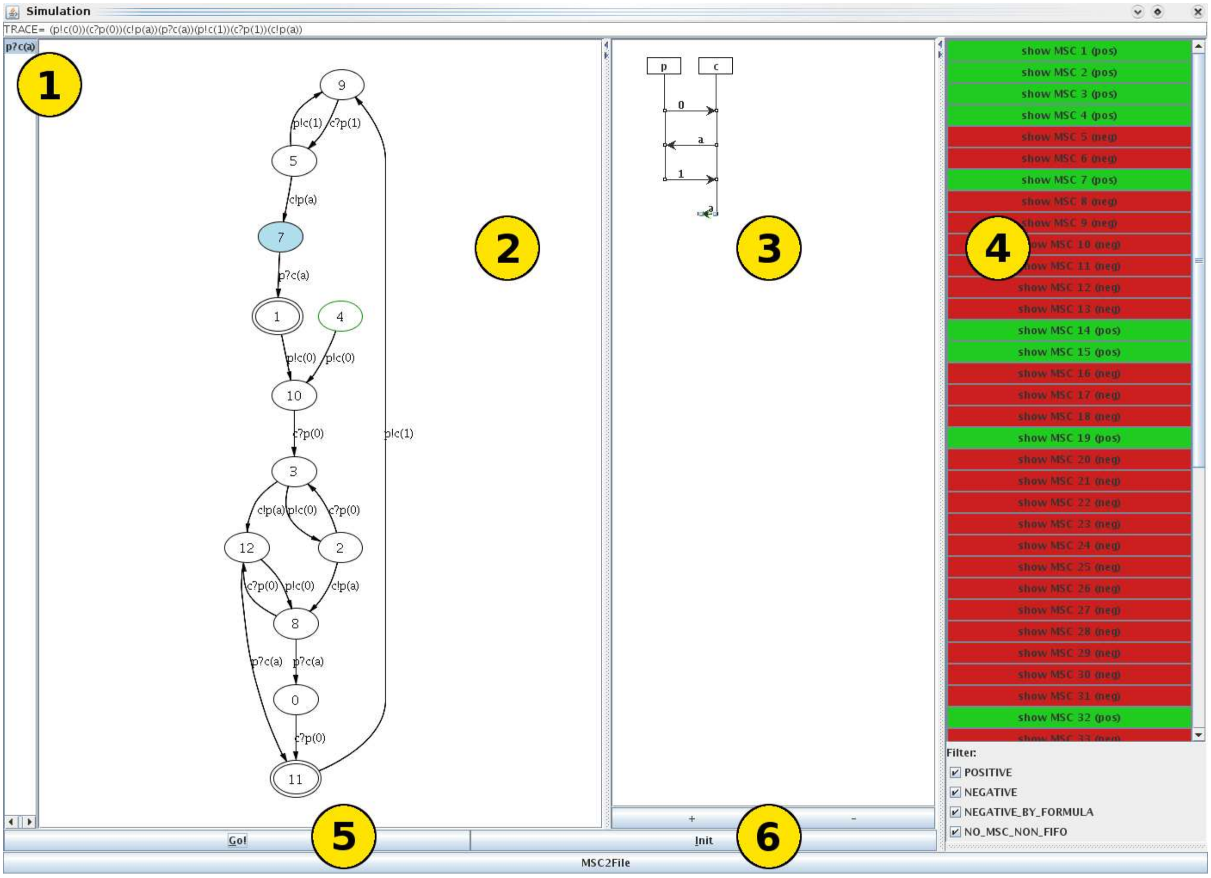
Formulae can be input into `Smyle` via the formula editor presented in Figure 9.7. When the user is asked to classify an MSC (1), he may add such formulae to the current formulae list (2). PDL formulae which fulfill the current MSC (1) are marked *green*, the others *red* (cf. left side of component (2)). Whenever the user chooses to “Add” a PDL formula, the editor appears (3) in which the formula can be specified. A menu (4) assists the user in correctly writing down the formula. Moreover, the editor supports syntax-error highlighting (cf. lower left corner of (4)) and counting (cf. (5)).

As we already explained in Section 7.4 this may lead to a considerable reduction in the number of user queries. Note, however, that the final system is not guaranteed to fulfill all specified patterns on all runs. Patters are currently only used to ease the specification task by reducing the number of user queries and not to ensure desired properties of the underlying system. This is because, at present, there are no model checkers for CFMs available.

Note that for arbitrary CFMs the model checking problem is undecidable. In theory, it is, however, possible—though involved—to check the classes of CFMs regarded in this thesis with respect to the patterns specified by the user [BKM07].

Simulating a Hypothesis

Whenever `Smyle` has a complete and consistent view of the current internal model, it presents a window (cf. Figure 9.8) for testing and simulating the derived system. Within this component, the user (now acting as *Oracle*) may execute action sequences to see how


Simulation
p?c(a)
1
9
p!c(1)
c?p(1)
5
c!p(a)
7
p?c(a)
1
4
p!c(0)
p!c(0)
10
c?p(0)
p!c(1)
3
c!p(a)
p!c(0)
c?p(0)
12
2
c?p(0)
p!c(0)
c!p(a)
8
p?c(a)
p?c(a)
0
c?p(0)
11
2
p
c
0
a
1
a
3
show MSC 1 (pos)
show MSC 2 (pos)
show MSC 3 (pos)
show MSC 4 (pos)
show MSC 5 (neg)
show MSC 6 (neg)
show MSC 7 (pos)
show MSC 8 (neg)
show MSC 9 (neg)
show MSC 10 (neg)
show MSC 11 (neg)
show MSC 12 (neg)
show MSC 13 (neg)
show MSC 14 (pos)
show MSC 15 (pos)
show MSC 16 (neg)
show MSC 17 (neg)
show MSC 18 (neg)
show MSC 19 (pos)
show MSC 20 (neg)
show MSC 21 (neg)
show MSC 22 (neg)
show MSC 23 (neg)
show MSC 24 (neg)
show MSC 25 (neg)
show MSC 26 (neg)
show MSC 27 (neg)
show MSC 28 (neg)
show MSC 29 (neg)
show MSC 30 (neg)
show MSC 31 (neg)
show MSC 32 (pos)
show MSC 33 (neg)
4
Go!
5
+
-
Init
6
MSC2File

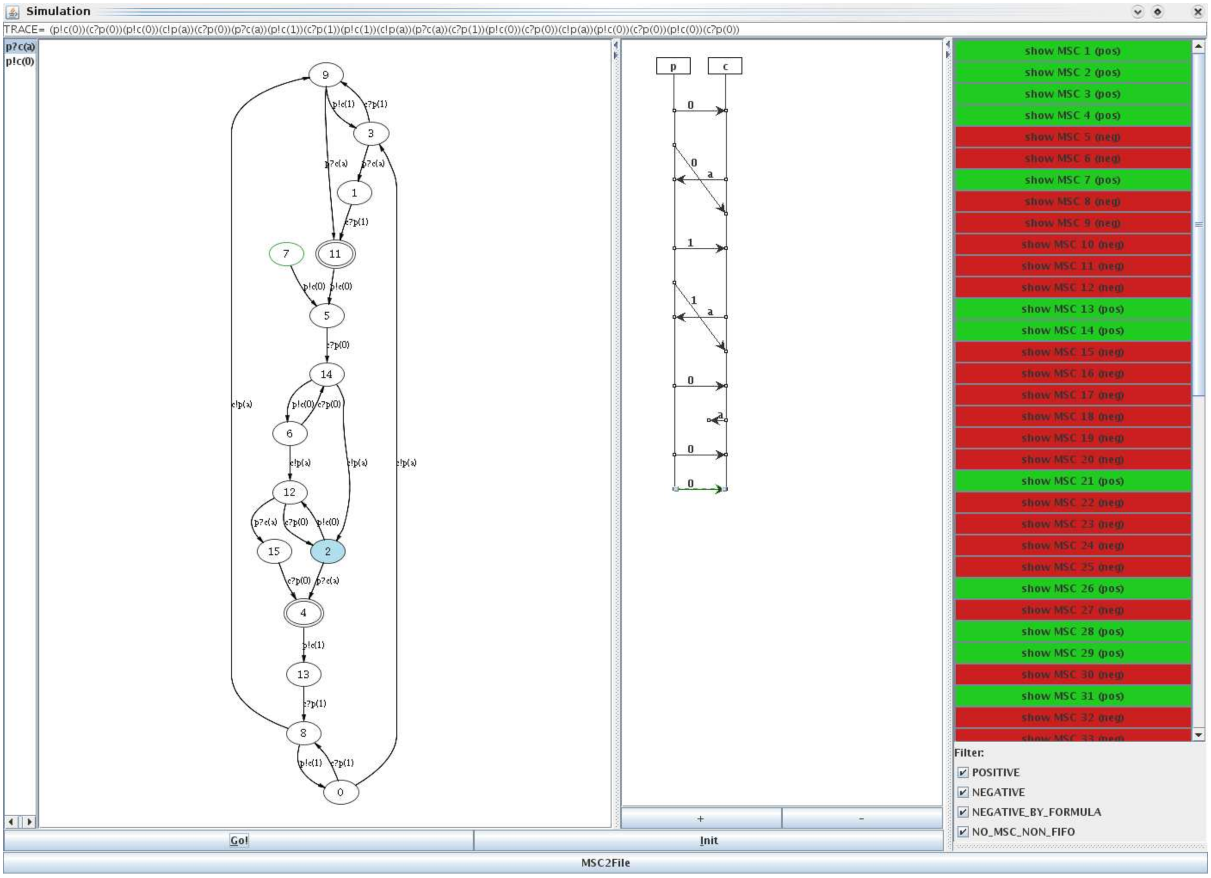
TRACE= (p!c(0))(c?p(0))(c!p(a))(p?c(a))(p!c(1))(c?p(1))(c!p(a))

Filter:
- ☑ POSITIVE
- ☑ NEGATIVE
- ☑ NEGATIVE_BY_FORMULA
- ☑ NO_MSC_NON_FIFO

Figure 9.8: Simulating an intermediate model in `Smyle`

the system behaves. To this end, in the upper left corner (cf. Figure 9.8 (1)) the simulation component lists all communication actions that are available in the current state of the hypothesis (in Figure 9.8 this is only receive action $?(p, c, a)$). If the user chose an action hitting the "*Go!*" button (5), the system evolves (2) from one state to another. Moreover, these actions are monitored, and the related (partial) scenario, depicted as MSC (prefix) in the middle of the frame (cf. Figure 9.8 (3)), is updated accordingly. Using button (6) resets the automaton and the current MSC prefix to their initial states. Of course, it may also be helpful to automatically derive sets of accepted or rejected MSCs, or to complete a user-specified MSC prefix to full MSCs. This might be valuable for the user to further understand the system under simulation. This feature is about to be integrated into the simulation component. The simulation component furthermore exhibits a panel that collects all MSCs classified (either by the user, by the learning setup or by PDL formulae) so far in the right panel (4). Each of the MSCs is represented by a button which, when pushed, presents the corresponding MSC in a new frame. Green buttons typify positively classified MSCs and red buttons negatively classified MSCs (either because of a user classification, or because a PDL formula matched). Moreover, filters at the bottom of panel (4) can be employed to restrict to certain properties (e.g., "show only the positively classified scenarios").

If, after an intensive simulation, there is no evidence for wrong or missing behavior (cf. Figure 9.9), the user will terminate the simulation session, and the concurrent system will be deduced. If, however, some illegal or missing behavior is detected, then the user can use the corresponding MSC as negative counterexample, or, respectively, extend a partial run


Simulation
p?c(a)
p!c(0)
9
p!c(1)
c?p(1)
3
p?c(a)
p?c(a)
1
c?p(1)
7
11
p!c(0)
p!c(0)
5
c?p(0)
14
c!p(a)
p!c(0)
c?p(0)
6
c!p(a)
c!p(a)
c!p(a)
12
p?c(a)
c?p(0)
p!c(0)
15
2
c?p(0)
p?c(a)
4
p!c(1)
13
c?p(1)
8
p!c(1)
c?p(1)
0
p
c
0
0
a
1
1
a
0
a
0
0
+
-
show MSC 1 (pos)
show MSC 2 (pos)
show MSC 3 (pos)
show MSC 4 (pos)
show MSC 5 (neg)
show MSC 6 (neg)
show MSC 7 (pos)
show MSC 8 (neg)
show MSC 9 (neg)
show MSC 10 (neg)
show MSC 11 (neg)
show MSC 12 (neg)
show MSC 13 (pos)
show MSC 14 (pos)
show MSC 15 (neg)
show MSC 16 (neg)
show MSC 17 (neg)
show MSC 18 (neg)
show MSC 19 (neg)
show MSC 20 (neg)
show MSC 21 (pos)
show MSC 22 (neg)
show MSC 23 (neg)
show MSC 24 (neg)
show MSC 25 (neg)
show MSC 26 (pos)
show MSC 27 (neg)
show MSC 28 (pos)
show MSC 29 (pos)
show MSC 30 (neg)
show MSC 31 (pos)
show MSC 32 (neg)
show MSC 33 (neg)
Go!
Init
MSC2File

TRACE= (p!c(0))(c?p(0))(p!c(0))(c!p(a))(c?p(0))(p?c(a))(p!c(1))(c?p(1))(p!c(1))(c!p(a))(p?c(a))(c?p(1))(p!c(0))(c?p(0))(c!p(a))(p!c(0))(c?p(0))(p!c(0))(c?p(0))

Filter:
☑ POSITIVE
☑ NEGATIVE
☑ NEGATIVE_BY_FORMULA
☑ NO_MSC_NON_FIFO

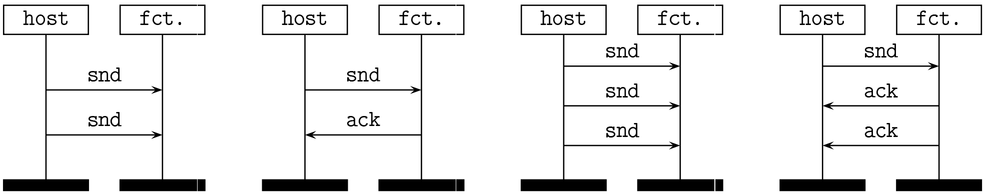
Figure 9.9: Simulating a final model (no evidence for wrong behavior is found)

to a missing scenario to obtain a counterexample which has to be classified as positive. An example is given in Figure 9.8 (3). As mentioned before, the partial scenario can—in this situation—only be extended by a receive action $?(p, c, a)$. But our protocol should be able to send another message with content 1 from p to c (assuming that we are about to infer the ABP). As this, however, is not possible, our hypothesis has to be extended. To this end, we may amplify the MSC prefix to a scenario that is not contained in the language of the hypothesis resulting in a positive counterexample. This single counterexample, possibly enriched by additional MSCs yielding a counterexample suite, is supplied to the learning chain, and the learning procedure continues as explained before until reaching the next consistent model. A short exemplifying video of this learning process can be downloaded from the tool's webpage [`Smyle`].

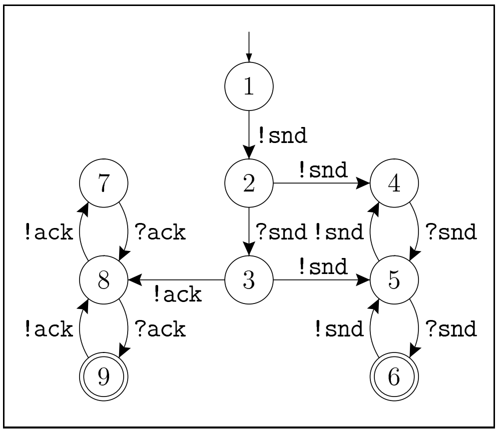
9.2 Case Studies

We applied `Smyle` to several small and mid-size case studies. Among them were a protocol being part of USB 1.1 mentioned in [Gen05], the *continuous update protocol* from [EMST03], the *negotiation protocol* from [EMST04], the well-known *alternating bit protocol* (cf. [Tan02] and Section 7.4), and two variants of the *leader election protocol* from [CR79]. Protocols, such as the ABP or leader election, are known to be error-prone whereas the automata generated by `Smyle` are guaranteed to be correct by construction (provided, of course, that the MSCs are specified correctly by the user).

In the following, we describe four of the protocols `Smyle` was applied to in more de-



(*i*) The four *positive* example MSCs `Smyle` was provided with



(*ii*) The inferred hypothesis DFA after 14 user queries and 200 membership queries

Figure 9.10: (*i*) Input MSCs and (*ii*) correct and complete hypothesis DFA

tail. The related learning statistics (i.e., number of membership-, user-, and equivalence queries), results without and with our partial-order learning approach, the size of the hypothesis, the table size and memory consumption, and information on the learning setup used to infer the protocols can be found in Table 9.1 on page 159.

To clarify the notations used in this table, let us briefly summarize the main terms:

(i) *Membership queries* are all queries asked by the implementation of Angluin's learner (i.e., the `libalf` library). They are mostly answered automatically by our procedure presented in Chapter 6.

(ii) Contained in the number of membership queries are the *user queries.* User queries are the inquiries (i.e., MSC pictures) that are displayed to the `Smyle` user and have to be answered manually (as long as we do not use PDL formulae to ease this task). To this end, the user must classify the presented MSCs as either positive or negative scenarios by pressing the *Accept* or *Reject* button, respectively.

(iii) Not included in the above numbers are *equivalence queries.* As Table 9.1 shows, equivalence queries are rather rare and have to be answered by testing and simulating the hypothesis. To this end, `Smyle` features an integrated simulation component. If the user is not satisfied with the hypothesis, she has to provide at least one counterexample MSC to the tool to revive the learning chain.

Part of the USB 1.1 protocol

The first protocol we regard, which is also mentioned in [Gen05], is part of the *USB 1.1 specification* [usb]. It describes two processes `host` and `function` (`fct.`) and their communication being initiated by the `host`. The first message sent from the `host` informs process `fct.` that the isochronous mode (USB distinguishes between three kinds of modes: isochronous, bulk, and setup) will be used and, moreover, whether the `fct.` has to play the role of the receiver or the transmitter of data.

To learn this protocol, we provided `Smyle` with the four positive scenarios depicted in Figure 9.10 (*i*). After answering another 14 user queries `Smyle` presented the correct and complete existentially 2-bounded ($\exists 2$) hypothesis automaton from Figure 9.10 (*ii*) which, indeed, describes the protocol we wanted to obtain. Applying our partial-order learning approach from Section 6.3, the number of membership queries could be lowered from 488 to 200, resulting in membership-query savings of 59.0% and memory savings of 46.5%.

Note that in Figure 9.10 (*ii*) we used shorthands for the communication actions because sending and receiving processes are clear from the context (e.g., `!ack` is an abbreviation for !(`fct.`, `host`, `ack`)).

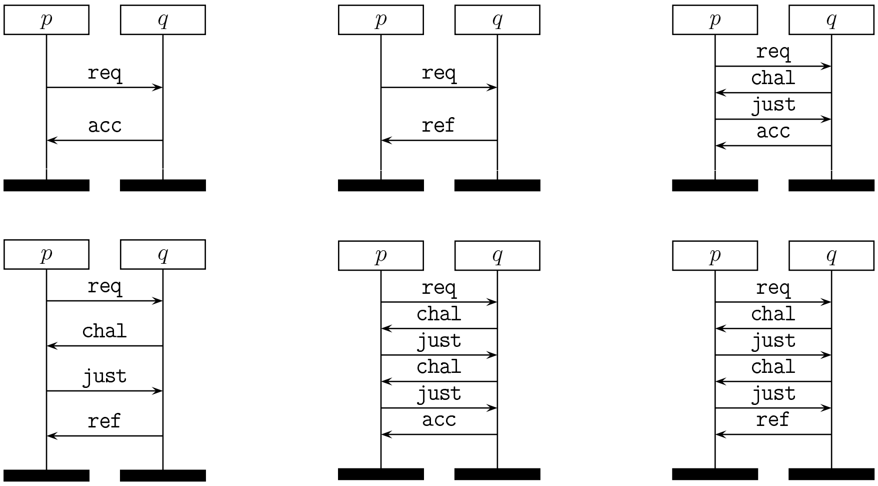
Note that the inferred minimal DFA of Figure 9.10(*ii*) has a local-choice problem as after the first send event either a phase of only sending and receiving `snd` messages from p to q or `ack` messages from q to p occurs. In such cases, an implementation might suffer from deadlocks. These local-choice problems can be detected using analysis tools such as `MSCan` [BKSS06]. In general, however, our approach cannot resolve such problems automatically. We can only guarantee to avoid deadlocks in the learning setups for deterministic $\forall B$-bounded deadlock-free and deterministic $\forall B$-bounded deadlock-free weak CFMs. In the example, the local-choice problem can be resolved by using synchronization messages.

Negotiation protocol

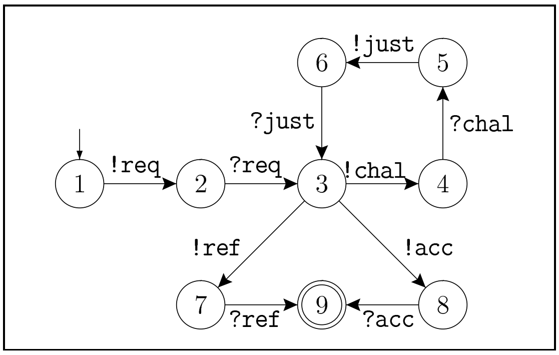
The *negotiation protocol* [EMST04] is a small protocol describing a negotiation dialog between two processes p and q where p is acting as a customer requesting some information, and server q owns this information. The server tries to discover whether or not it may provide this information to p. Client p sends a `request` (`req`) to server q. The server may either directly `accept` (`acc`) or `refuse` (`ref`) the client's request, or enter a `challenge-justify` (`chal` and `just`, respectively) phase in which it inquires additional data about the client. As long as the server is not satisfied with the information provided by the client he stays in this phase. Once the server collected enough information it decides whether to `accept` or `refuse` the client's initial request.

For this protocol, `Smyle` was fed with the six positive MSCs from Figure 9.11 (*i*) as input and learned the automaton after 31 user queries. The learning library `libalf` asked the *Assistant* 1,179 membership queries and provided the user with the final result depicted in Figure 9.11 (*ii*). This protocol is indeed the desired existentially 1-bounded ($\exists 1$) version of the functionality we had in mind. Concerning the partial-order learning optimization, we were able to reduce the number of membership queries from 1,179 to 432 yielding a decrease of 63.4% and achieved memory savings of 54.0%.

As in the previous example, we use shorthands for the communication primitives.


p
q
req
acc
p
q
req
ref
p
q
req
chal
just
acc
p
q
req
chal
just
ref
p
q
req
chal
just
chal
just
acc
p
q
req
chal
just
chal
just
ref

(*i*) The six *positive* example MSCs `Smyle` was provided with


!just
6
5
?just
?chal
1
!req
2
?req
3
!chal
4
!ref
!acc
7
?ref
9
?acc
8

(*ii*) The inferred hypothesis DFA after 31 user queries and 432 membership queries

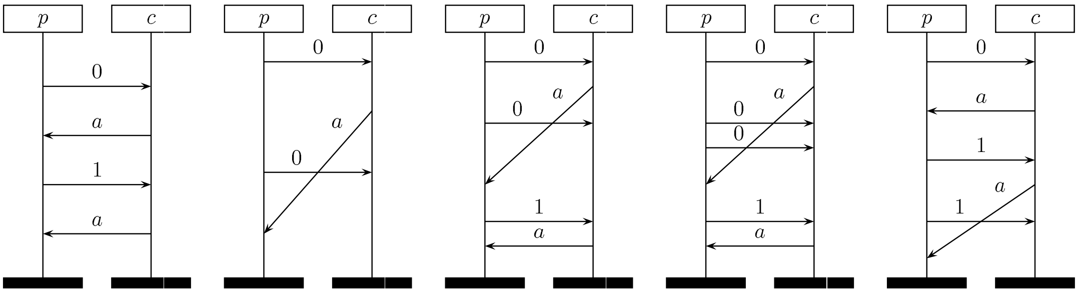
Figure 9.11: (*i*) Input MSCs and (*ii*) correct and complete hypothesis DFA

Alternating bit protocol (ABP)

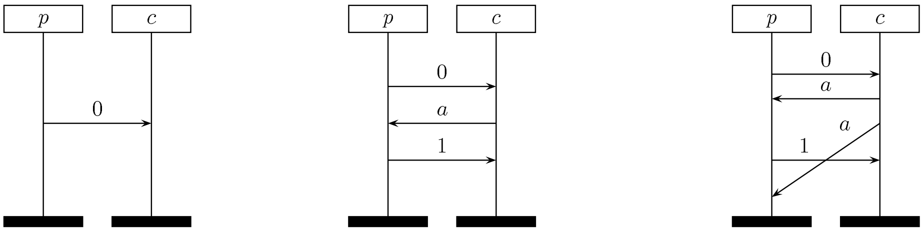
For the sake of completeness, we briefly describe the ABP introduced in Section 7.4. Its main goal [Tan02] is to ensure the reliability of data transmission through an unreliable FIFO channel, i.e., data loss as well as data duplication are possible. There are two processes participating in the communication, namely the *producer* p and the *consumer* c. Moreover, the channel between producer and consumer is lossy whereas the other direction is known to be reliable.

The protocol works as follows: initially a bit `b` is set to 0. Process p keeps sending the value of `b` until it receives an acknowledgment message a from process c. After receiving such an acknowledgment, process p locally inverts the value of `b` and starts sending the new value until the next a message is received from c. The communication can terminate after any a (but at least one) that was received at p. Note that there is no reason to distinguish between the acknowledgment messages because the channel (c, p) is assumed to be faultless.

The automaton we tried to learn was specified as existentially 1-bounded ($\exists 1$). Feeding


p
c
0
a
1
a
0
0
1
a

(*i*) The five *positive* example MSCs `Smyle` initially was provided with to infer the ABP

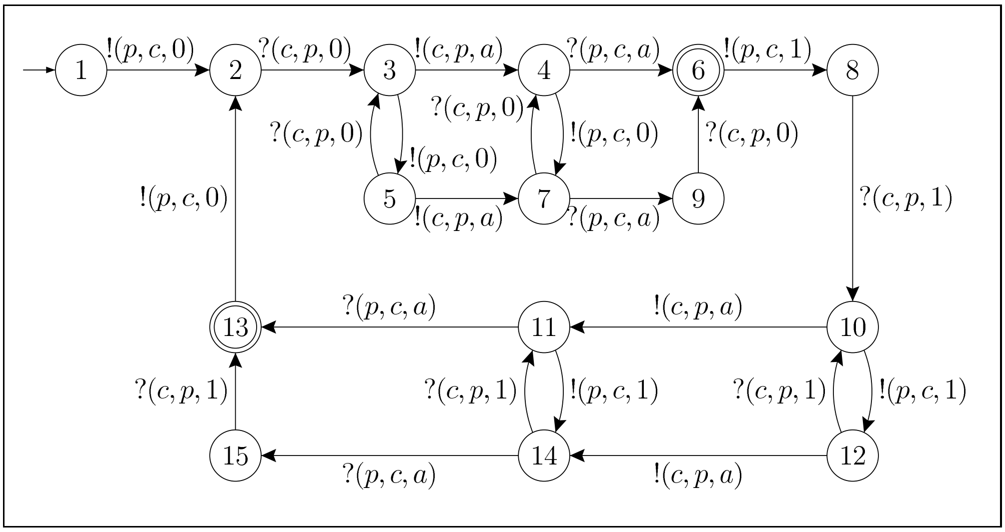


no final acknowledgment

no final acknowledgment

two subsequent acknowledgments on q

(*ii*) Some negative MSCs classified during the learning phase, and the reasons for their negative classification


1
2
3
4
6
8
5
7
9
10
11
13
12
14
15
!(p, c, 0)
?(c, p, 0)
!(c, p, a)
?(p, c, a)
!(p, c, 1)
?(c, p, 0)
!(p, c, 0)
?(c, p, 1)
!(c, p, a)
?(p, c, 1)
?(c, p, 1)
!(p, c, 1)

(*iii*) The inferred hypothesis DFA after 64 user queries and 697 membership queries

Figure 9.12: (*i*) Input MSCs (*ii*) three negative MSCs, and (*iii*) correct and complete hypothesis DFA

`Smyle` with the 5 positive MSCs from Figure 9.12 (*i*) our tool provided us with the correct hypothesis depicted in Figure 9.12 (*iii*) (the corresponding CFM can be found in Figure 7.10 on page 129) after 64 user queries. Some negative examples that occurred during the protocol inference are given in Figure 9.12 (*ii*). Note that the *Assistant* was asked 2,286 membership queries. This number could be lowered by 69.5% to 697 membership queries, employing partial-order learning. The memory consumption could be reduced by 60.9%, accordingly.

Additionally, `Smyle` learned the ABP for existentially 2-bounded (∃2) and existentially 3-bounded (∃3) learning setups. The statistical results are similar and can be found in Table 9.1 on page 159.

Leader election protocol

Consider a *leader election protocol* in a unidirectional ring from [CR79]. Leader election plays an important role in many distributed applications. In a network of identical (up to their unique process id) communicating units one often uses a leading entity, here called `leader`, (owning a unique *leader token*) to control the behavior of the others. However, a problem arises if due to communication failures or other possible problems, the leader token gets lost. The goal of leader election protocols is then to select a unique leader out of the network of computers or processes, etc.

In general, the protocol works as follows: one process starts sending its `pid` to its clockwise neighbor who compares the value of the received `pid` with its own. If its own `pid` has got a higher value, it sends its `pid` to the next process. However, if its `pid` exceeds the received one, the current process just forwards the received `pid` to its clockwise neighbor.

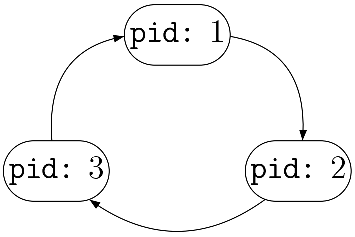
In case both values are equal, a leader has been found and the current process declares itself the new leader by broadcasting message m_`leader`.

Due to the high amount of concurrency, the protocol version we learn in this example is restricted to three processes (cf. Figure 9.13 (*i*)) and to sending only one message at each point in time.

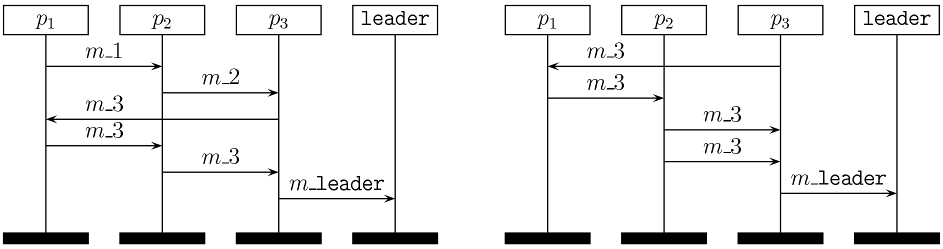
`Smyle` learned two versions of the protocol. The first one (v_1) (cf. Table 9.1) is only capable of performing one round of electing a leader whereas the second version (v_2) (cf. Figure 9.13 and Table 9.1) can execute arbitrary many elections consecutively.

`Smyle` learned version v_1 of the leader election protocol obtaining three input MSCs. It displayed the correct hypothesis after 43 user queries and 3,612 membership queries. Applying the partial-order learning approach, we were able to lower this number by 75.1% to 900 membership queries and achieve memory savings of 70.8%. For the more elaborate version (v_2), `Smyle` received six input MSCs (e.g., the first one of Figure 9.13 (*ii*)) and returned the correct hypothesis depicted in Figure 9.13 (*iii*) after 196 user queries (cf. Figure 9.13 (*ii*) for a negative scenario) and 14,704 membership queries. Among the negative user queries was, for example, the right MSC from Figure 9.13 (*ii*). Here, too, the number of membership queries could be substantially decreased by 53.3% yielding a total number of 6,864 membership queries when using partial-order learning, and, moreover, memory savings of 47.3%.

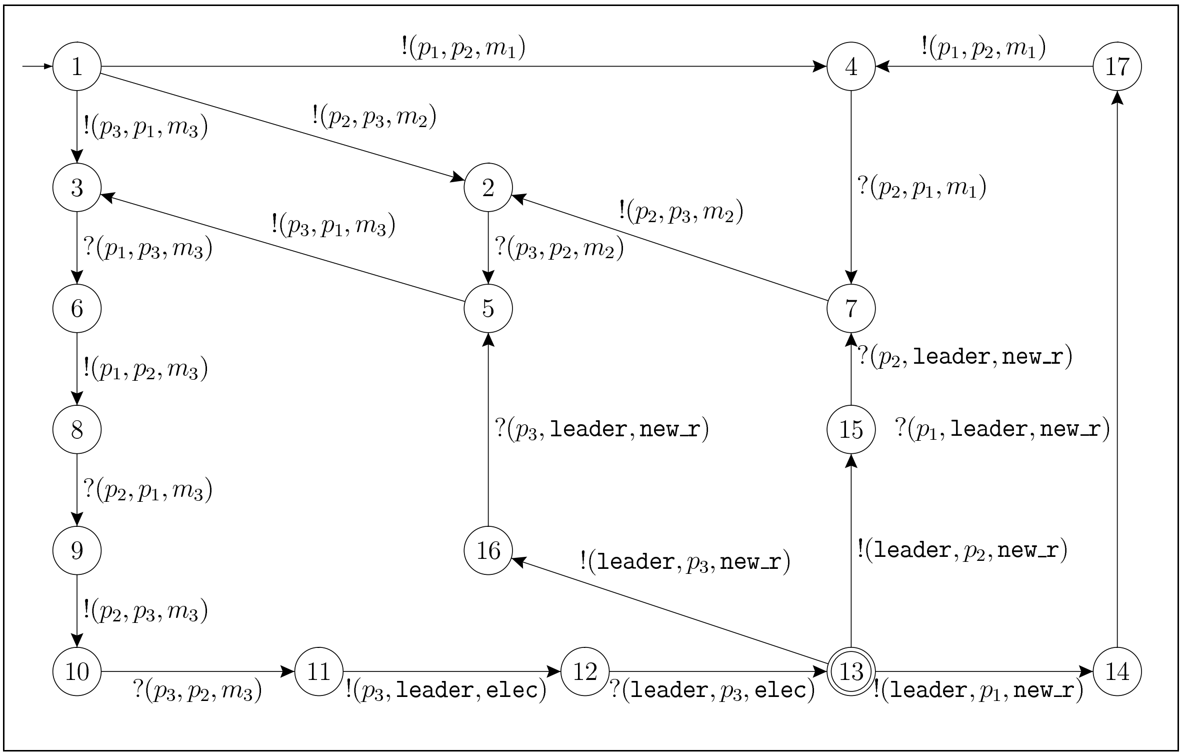
A note on using PDL formulae: As stated before, the use of PDL to ease the user's task of classifying scenarios can substantially reduce the number of user queries. As, however, different formulae yield totally different results, it is difficult to generate statistics


pid: 1
pid: 2
pid: 3

(i) Unidirectional ring with three identical processes but unique process ids (pids)


p_1
p_2
p_3
leader
m_1
m_2
m_3
m_3
m_3
$m_$leader
p_1
p_2
p_3
leader
m_3
m_3
m_3
m_3
$m_$leader

(ii) One positive and one negative scenario for the leader election protocol


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
$!(p_1, p_2, m_1)$
$!(p_1, p_2, m_1)$
$!(p_3, p_1, m_3)$
$!(p_2, p_3, m_2)$
$?(p_1, p_3, m_3)$
$!(p_3, p_1, m_3)$
$?(p_3, p_2, m_2)$
$!(p_2, p_3, m_2)$
$?(p_2, p_1, m_1)$
$!(p_1, p_2, m_3)$
$?(p_2, \texttt{leader}, \texttt{new\_r})$
$?(p_3, \texttt{leader}, \texttt{new\_r})$
$?(p_1, \texttt{leader}, \texttt{new\_r})$
$?(p_2, p_1, m_3)$
$!(\texttt{leader}, p_3, \texttt{new\_r})$
$!(\texttt{leader}, p_2, \texttt{new\_r})$
$!(p_2, p_3, m_3)$
$?(p_3, p_2, m_3)$
$!(p_3, \texttt{leader}, \texttt{elec})$
$?(\texttt{leader}, p_3, \texttt{elec})$
$!(\texttt{leader}, p_1, \texttt{new\_r})$

(iii) The inferred hypothesis DFA $\mathcal{H}$ after 196 user queries (and 6,864 membership queries)

Figure 9.13: Automaton for the *leader election protocol* (v_2) in a unidirectional ring. (i) Ring for 3 processes, (ii) Some input MSCs, (iii) hypothesis DFA

	#membership queries			#user	#equiv.		#rows in table			learning
Protocol	**w.o. POL**	**w. POL**	**savings**	**queries**	**queries**	$\lvert\mathcal{H}\rvert$	**w.o. POL**	**w. POL**	**reduction**	**setup**
part of *USB 1.1*	488	200	59.0%	14	1	9	61	26	57.4%	∃2
continuous update	712	264	62.9%	21	1	8	89	34	61.8%	∃1
negotiation	1,179	432	63.4%	31	1	9	131	49	62.6%	∃1
alternating bit	2,286	697	69.5%	64	2	15	127	42	66.9%	∃1
alternating bit	14,432	4,557	68.4%	158	2	25	451	131	71.0%	∃2
alternating bit	55,131	19,252	65.1%	407	2	37	799	222	72.2%	∃3
leader elec. (v_1) (1 rnd)	3,612	900	75.1%	43	1	13	301	76	74.8%	∀
leader elec. (v_2) ($\geq$ 1 rnds)	14,704	6,864	53.3%	196	2	17	919	430	53.2%	∀

Table 9.1: Statistical results of the case studies

concerning this reduction. Moreover, depending on the application domain and the expertise of the `Smyle` user, the number of autonomously classified scenarios may also vary drastically. Note, though, that we gave a small example for this reduction in Section 7.4.

9.3 Implementation Details

To provide a platform-independent application, `Smyle` is written in `Java` (Version 1.6) and, originally, used the `LearnLib` library [RSB05, RS06] developed at the university of Dortmund, which implements Angluin's learning algorithm L*. As we previously mentioned, while employing the `LearnLib` library we faced several shortcomings. The first problem was the unavailability of source code. As we invented several optimizations concerning the storage of data in Angluin's table, we were in great need of extending the library. A second and more severe drawback was the mandatory Internet connection to the university of Dortmund where the `LearnLib` server is located. To use the `LearnLib` it is required to connect to the password protected server. After logging in, it will send membership- and equivalence queries via the Internet to the current `Smyle` instance which in turn answers and returns them to the `LearnLib` server. As soon as there was no Internet connection, `Smyle` was, unfortunately, not applicable anymore. Being confronted with such inconveniences, we decided to initiate a new project called `libalf` implementing prominent learning algorithms (e.g., Angluin's L*, our nondeterministic learning approach NL*, Biermann, etc.) and optimizations thereof (e.g., learning congruence-closed languages). We already reported about this library in Chapter 8 of this dissertation. It is now integrated into `Smyle` and performs very well. All examples presented in this chapter have been derived using the dispatcher component of `libalf`. We are currently extending `Smyle` to support communication with `libalf` via the JNI. This will slightly speed up the learning task.

In the rest of this section, we will report on implementation details like important algorithms and architecture, and give a small overview over `Smyle`'s history and future.

9.3.1 Visualization

For visualization purposes `Smyle` makes use of two freely available libraries called `JGraph`[1] [JGr] and `Grappa`[2] [AT&T]. `JGraph` is a powerful open source graph library which is easily integrable into `Java` Swing components. Within `Smyle`, we use it for visualizing MSCs and MSC components, as well for displaying user queries in terms of MSCs (cf., e.g.,

---

[1] JGraph: `http://www.jgraph.com/`

[2] Grappa: `http://www.research.att.com/~john/Grappa/`

Figures 9.3 (4) and 9.5), as for the MSC editor (cf. Figure 9.6). The second library is developed by AT&T Labs - Research. The name `Grappa` is an acronym for **GRAP**h **PA**ckage. This library provides means for visualizing graphs in a `Dot`-style manner and uses the *Graphviz* program `Dot` for layouting the graphs. Inside `Smyle`, it visualizes the intermediate hypothesis automata, as well as the component for simulating the final implementation model. Therefore, it is recommended to either have a version of `Dot` installed on the system where `Smyle` is executed, or to have an Internet connection such that the `Grappa` library can connect to a dedicated server from which it is able to retrieve the necessary layout information.

9.3.2 Algorithms

We implemented several algorithms indispensable for the correct functioning of our application. At first, as the learning library will always query words and not partial orders (MSCs) directly, an algorithm for checking words for well-formedness (cf. Definition 6.1.3 on page 88) was essential. Being able to detect whether a word was well-formed, a method was required to transform the linearization into an MSC which, in turn, could be converted into a picture that subsequently would be displayed to the user. As an MSC usually represents a whole set of linearizations, we moreover needed an algorithm that is capable of deriving all linearizations of an MSC iteratively, and for efficiently checking the membership of a linearization with respect to the partial order that the MSC represents. For computing these linearizations we used the algorithm described in [VR81] which takes as input a partial order (e.g., an MSC), and calculates all its linear extensions (i.e., linearizations). The algorithm runs in $\mathcal{O}(n \cdot e(\mathscr{M}))$ time, where n is the number of elements of the partial order $\mathscr{M}$ and $e(\mathscr{M}) = |E(\mathscr{M})|$ is the number of linear extensions of $\mathscr{M}$. We slightly changed the structure of the algorithm to get an iterator over the set of linearizations of an MSC. Given an input partial order, the tool creates an iterator which stepwise returns linearizations whenever the user requests one.

An important improvement regarding the reduction of the number of user queries was to employ PDL patterns. To this end, we needed to implement a PDL parser and a small model checker which takes an MSC and a PDL formula as input and tries to verify or falsify the formula with respect to the MSC in question. The actual algorithm for checking local formulae (with and without path expressions) is given in Appendix C.

For creating test suites or simulation sets, i.e., set of words or MSCs accepted or rejected by the hypothesis automata, we employed different search methods, such as the well-known *depth-first* and *breadth-first* search methods but also random search, yielding a greater variety of "diverse" MSCs. These methods are integrated into the simulation component and the test case generation application `Style`, which will be discussed at the end of this chapter.

The complete and efficient implementation of Angluin's L^* algorithm was realized within our learning library `libalf`, which was described previously in Chapter 8. In this library, we also integrated our partial-order approach for reducing memory consumption and the number of membership queries.

9.3.3 Parsers

`Smyle` employs two parsers for different purposes. The external `MSC2000` parser [Neu] is used for parsing the input MSC documents. Though `Smyle` only uses basic MSCs, `MSC2000` is capable of parsing a much larger part of the MSC ITU standard [ITU99]. The second parser was developed within the `Smyle` project. It is a PDL parser which can process all kinds of PDL formulae. Together with the small integrated model checker, it provides means to reduce the number of queries that are presented to the user during the learning phase.

9.3.4 Learning Architecture

The `Smyle` tool is capable of learning CFMs from following classes of CFMs: universally bounded, existentially B-bounded, and deterministic universally bounded deadlock-free weak CFMs.

The learning framework contains the following three main components as depicted in Figure 9.2 on page 145: the *Learner*, the *Assistant* and the *University*.

(i) The *Learner* executes the main learning loop in which it answers the questions from the learning algorithm (implemented in `libalf`). Most of these queries are answered autonomously. Only in case the *Assistant* is unable to answer these queries, they are forwarded to the user as user queries. To this end, it first has to establish a connection to the learning library `libalf` and to create a learning instance of L*. Communication with the learning library is then performed via a clear network interface or JNI.

(ii) The *Assistant* keeps track of membership queries that have not yet been asked. I.e., for unasked linearizations of already classified MSCs, it checks words for well-formedness, B-boundedness, and the FIFO property, as well as the corresponding MSCs for their language type and with regard to user-specified patterns. As such, the *Assistant* acts as a kind of filter restricting the questions of L* to the domain specified by the learning setup. To be able to answer equivalent membership queries equally, it stores the set of already (user- and autonomously-) classified MSCs. Moreover, it can provide counterexamples if the current local hypothesis, i.e., a hypothesis not yet presented to the user, is not in conformance with the learning setup. This way, the user is only bothered with interesting user- and equivalence queries.

(iii) The *University*, representing the interface between the graphical user interface and the learning components *Learner* and *Assistant*. It stores the properties of the learning setup, initiates the learning loop executed by the *Learner*, and handles the communication between *Learner*, *Assistant*, and the actual `Smyle` user who provides feedback via the GUI component.

9.3.5 Package Architecture

As depicted in Figure 9.14, `Smyle` consists of seven main components or packages: the graphical user interface (GUI), one package for learning components, one for graph components, one package for MSC components, a package for employing the PDL logic, one comprising the MSC editor functionality, and an interface to the learning library `libalf`. The functionality of these components is briefly sketched in the following.

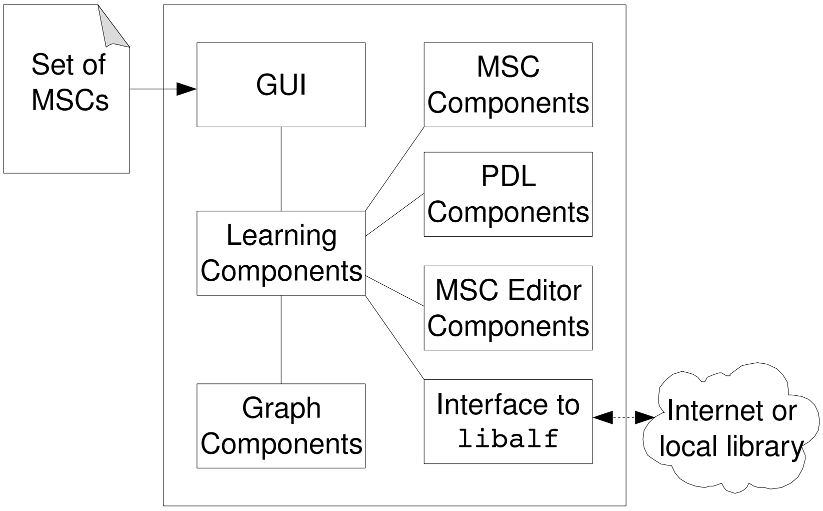


Figure 9.14: `Smyle`'s architecture: component overview

Graphical User Interface: The graphical user interface provides graphical means for improving and simplifying the user-computer interaction. It is based on `Java` Swing. An exemplifying screen shot was given in Figure 9.3 on page 146. The GUI is divided into three main components (1)-(3). Component (1) contains information about the current learning job (e.g., the specification of the learning setup, the input MSCs loaded from the file system, and autonomously derived counterexamples). The current local hypothesis is depicted on the right side of the window (component (2)). An output of the current hypothesis is given in `Dot` syntax in component (3), which is located at the bottom of components (1) and (2).

Learning Components: The tasks of the learning components are manifold. They contain important functionality for efficient partial-order treatment, harbor the simulator and the test case generator, which can be applied to the learned model, and comprise the *Learner*, the *Assistant*, and the *University* which acts as mediator between the components of this package and the other packages as shown in Figure 9.2.

Graph Components: The graph components package includes functionality for checking MSC behavior (e.g., the FIFO property) and the consistency of the implementation models. Moreover, it contains data structures for internal use (e.g., representing partial orders, etc.).

MSC Components: This package contains the MSC2000 parser [Neu] for handling MSC documents according to the ITU Z.120 standard [ITU99]. It provides the classes for representing the internal MSC objects.

PDL Components: The PDL components contain methods to parse PDL formulae and to model check these patterns with respect to a given MSC.

MSC-Editor Components: The MSC editor components feature the implementation of an integrated MSC editor, which is able to load, store, and alter basic MSCs. Moreover, the created MSCs can be exported to LaTeX and the `fig` format and, thus, can be converted to all other prevalent graphical formats (e.g., eps, pdf, jpeg) using available tools.

Interface to `libalf`: This package includes an interface to our learning library `libalf`, which implements Angluin's algorithm L* [Ang87a] and many others described in Chapter 3. A preliminary version of `Smyle` made use of an interface by courtesy of the *Fachbereich Informatik, Lehrstuhl 5* of the University of Dortmund.

9.3.6 `Smyle`: Past, Present and Future

`Smyle` is continuously being developed since autumn 2006. It is currently available in the stable version `0.3`, and contains approximately 24,000 SLoC[3] (not counting the parser component `MSC2000`, the learning library `libalf` and the other external libraries), and involves six people for software design and programming. Its main integrated features are summarized below:

- The following three language types are learnable in terms of CFMs: *universally bounded, existentially B-bounded, deterministic universally bounded deadlock-free weak* CFMs (cf. Chapter 6).
- The optimization of *partial-order learning* is included to conserve memory and lower the number of membership queries (cf. Chapter 5 and Section 6.3).
- A full *PDL parser*, *(textual) editor*, and *model checker* are integrated into `Smyle` (cf. Subsection 7.1).
- An embedded *simulation component* gives additional information and interesting insights about the correctness of the derived hypotheses.
- An integrated *test case generation component* (called `Style`, cf. Subsection 9.4.3) can derive large-size test suites either in terms of MSCs or system traces.
- Optionally, we developed an *MSC editor*, which is still in a prototypical state. Nevertheless, it can be used to graphically specify basic MSCs which can serve as input to `Smyle`.

For future work, it remains to extend the formula editor, the simulation component, and the MSC editor. The realization of the following ideas would be of great value as they will further ease the user's classification tasks:

Formula editor: A great achievement would be to have means for graphically specifying PDL formulae in some way. A first step into this direction would be to guide the user via a graphical formula editor where she clicks her formulae step by step. This helper is currently being integrated. Selecting parts of the formula might entail the tool to highlight the satisfying events or event sequences in the MSC. Most comfortable would be an extension where the user is able to annotate events, paths of events and so on in the currently displayed MSC, and `Smyle` (semi-) automatically infers a PDL formula satisfying the user specification. This may then be adjusted by the user and finally be submitted to `Smyle`.

---

[3] SLoC (source lines of code) calcultaed using `sloccount`

Simulation component: Some helpful extensions might turn the simulation component into an even more powerful support. As mentioned earlier, there are no dedicated model checkers for CFMs. But one could employ, or even integrate analysis applications like `MSCan` [BKSS06], for checking the resulting system for potential deficiencies like race conditions [EGP07], non-local choice, process divergence [BAL97], etc. Moreover, intermediate hypothesis automata (DFA) could be model checked by existing model checkers. Additionally, support of automatically deriving simulation sets (i.e., sets of MSCs which are accepted/rejected by the current hypothesis) will soon be integrated. This way, the user will get a more elaborate overview and gain more intuition about the underlying implementation model.

MSC editor: At present, the MSC editor is in a prototypical state. It will have to be upgraded to become a stable application, which can permanently be integrated into `Smyle`. Currently, it supports all basic MSC components used in `Smyle` but is not completely integrated, yet. Furthermore, several components from the MSC standard are still missing and need to be added. Afterwards, the MSC editor will be available within `Smyle` and, moreover, as a stand-alone version on the `Smyle` website.

9.4 Further Areas of Application

In addition to the application areas mentioned so far, i.e., learning regular CFM languages (cf. Chapter 6) and embedding `Smyle` into a software development lifecycle model (cf. Chapter 7), in the following, we propose some supplementary ideas and situations in which `Smyle` could be of help.

9.4.1 Analysis of Implementability

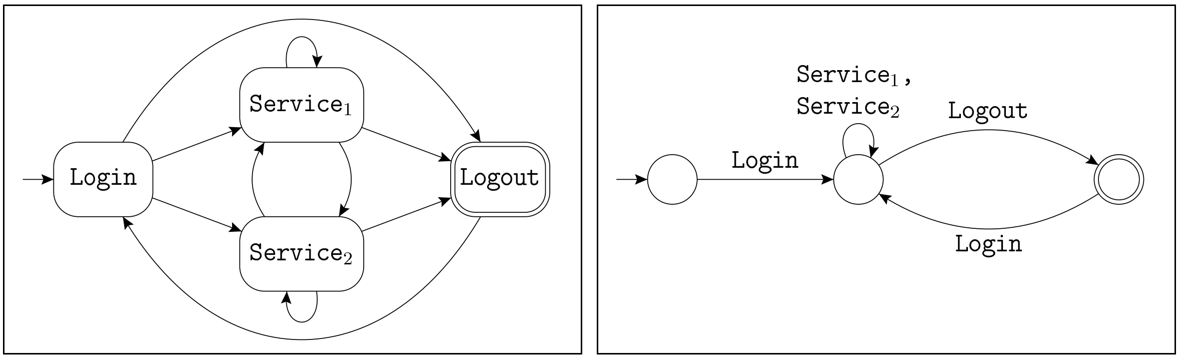
As `Smyle` is capable of learning regular CFM languages, and thus distributed system models, it could be used as analysis tool for distributed systems. Given a regular system description in terms of an automaton $\mathcal{A}$ (a DFA of minimal size) acting as teacher and oracle, and a concrete learning setup $\mathcal{S}$, `Smyle` could try to learn $\mathcal{A}$ with regard to $\mathcal{S}$ and thereby check whether $\mathcal{A}$ is implementable given the bound restrictions specified in the learning setup. Concrete questions that could arise are:

(i) Given a system with bound $B \in \mathbb{N}$, how does a realization with bound $k \in \mathbb{N}$ and $k < B$ look like (if it exists)?

(ii) Is there a possibility to realize a given system description with buffer size $B \in \mathbb{N}$?

9.4.2 Hierarchical Learning

An interesting idea arises when a distributed system can be divided into several subcomponents. Usually, for large systems this is the case. So, given a system $\mathcal{S}$ that consists of n subcomponents $C_1, \ldots, C_n$, the underlying communicating systems could be inferred using `Smyle`, yielding CFMs $\mathcal{A}_{C_1}, \ldots, \mathcal{A}_{C_n}$. Now, on a higher level of abstraction, the interaction between these components could also be learned using `Smyle`. The learning algorithm operates over the alphabet $\Sigma = \{1, \ldots, n\}$, where letter $i \in \{1, \ldots, n\}$ represents the execution of subcomponent C_i. The word "1 3 1 2", for example, would correspond to the sequential execution of CFMs $\mathcal{A}_{C_1}$, $\mathcal{A}_{C_3}$, $\mathcal{A}_{C_1}$, and $\mathcal{A}_{C_2}$. Thus, we learn an automaton $\mathcal{A}_{\text{global}}$ (in fact a DFA), whose edges are labeled with CFMs. The CFMs can then be connected according to the transition relation of $\mathcal{A}_{\text{global}}$, yielding the complete design model for system $\mathcal{S}$. Depending on the application, we could use *weak* or *strong* sequencing, respectively, between any two nodes of the global system, meaning that either each process may independently enter the next CFM after successfully executing his current CFM obligations (weak sequencing), or—using some kind of synchronization barrier—all processes together move to the next CFM after completely executing the current CFM (strong sequencing).



(a) Platform description with two services (b) Inferred automaton $\mathcal{A}_{\text{global}}$

Figure 9.15: Hierarchical learning example featuring two services

Imagine, for example, a community platform where people can login, are allowed to perform different actions or utilize different services while they are online, and finally logoff again. Such a system can, on a high level of abstraction, be divided into the components *Login*, *Service*$_1$, …, *Service*$_n$, and *Logout*. For each of these components, we infer a CFM employing `Smyle`. In a second step, the communication structure between these derived components is being inferred, again using `Smyle`. After a *Login*, for example, we can always initiate a *Logout* and return to the *Login* screen again, or perform some services in arbitrary order and then eventually *Logout*. In Figure 9.15(a) such a system description is given in terms of a transition system for two services `Service`$_1$ and `Service`$_2$. On the right side (cf. Figure 9.15(b)), the learned global automaton $\mathcal{A}_{\text{global}}$ is presented. Note that its edges are now labeled with CFMs called `Login`, `Logout`, `Service`$_1$, and `Service`$_2$ that were inferred in the first step of the hierarchical learning approach.

If applicable, employing high-level learning will facilitate learning to a great extend. The learning tasks are much clearer than when performing one global inference task for the whole system at once.

9.4.3 `Style`: The `Smyle` Test Case Generator

Another area of application arises from the field of *model-based testing* [BJK$^+$05]. Usually test cases are directly tailored to the system under test (SUT, for short). In model-based testing, a system model at a certain degree of abstraction is generated on basis of so-called *test requirements*, and test cases are derived from it. As these test cases now also are at a different level of abstraction than the concrete SUT, these abstract test cases have to be transformed into executable test cases (cf. Figure 9.16).

There are many different ways of test case generation for model-based testing. To name only two, e.g., *constraint programming* and *model checking* are employed.

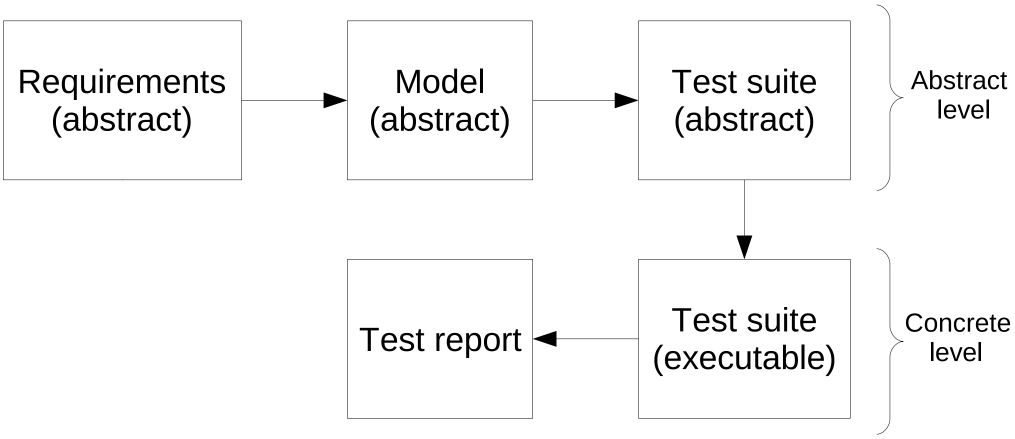


Figure 9.16: The model-based testing approach in general

In the constraint programming approach, the SUT is described by a set of constraints over certain variables. This set can be seen as a constraint satisfaction problem which—as described in the Biermann learning algorithm (cf. Subsection 3.2.1)—can be transformed into a satisfiability problem over Boolean variables, and be solved using existing SAT solvers yielding an abstract test case for the SUT.

A second strategy is to use model checking [CGP99, BK08]. Model checking in its essence tries to verify or falsify logical formulas on basis of a given labeled transition system. In the setting of model-based testing—being provided with an abstract model and a property to test—the model checking approach can derive paths to states which satisfy or disprove the property at hand. These witnesses or counterexamples then serve as abstract positive and negative test cases, respectively.

Often, the abstract model is seen as a finite-state automaton or transition system over states representing the underlying SUT. Then, test cases can be derived by applying different search methods (e.g., depth-, or breadth-first search, or random search) to the model searching for accepting, non-accepting, or non-present paths. These test cases are gathered in an abstract test suite.

We now want to describe a third approach in the direction of [HNS03] which employs the learning techniques described in the previous chapters. The so-called `Style` (i.e., the `Smyle` test case generator, where `Style` is an acronym for *Synthesizing Test cases bY Learning from Examples*) approach is an economical way to create or rebuild an abstract model of a system—even if the former model was not preserved for later use, or got lost between two development phases—and to produce large-size test suites, afterwards. As the first part of this approach makes use of our `Smyle` tool described previously, an evolution of the abstract model following the `SMA` lifecycle model (cf. Chapter 7) is also possible as soon as the underlying system is extended, or enters a new iteration of the software development cycle.

Usually, for testing a system, we are not directly given an abstract model of the SUT for performing model-based testing. Thus, we will learn this model given a set of test requirements. As depicted in Figure 9.17, we transform the abstract requirements into an abstract model using `Smyle`. If the model was created, the integrated test case generator `Style` can derive large test suites consisting of MSCs or system traces.

The test case generator's graphical user interface is depicted in Figure 9.18. Currently it supports breadth-first and depth-first search methods and some random search imple-

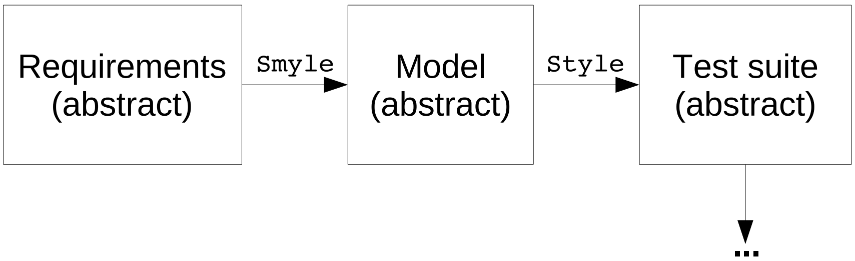
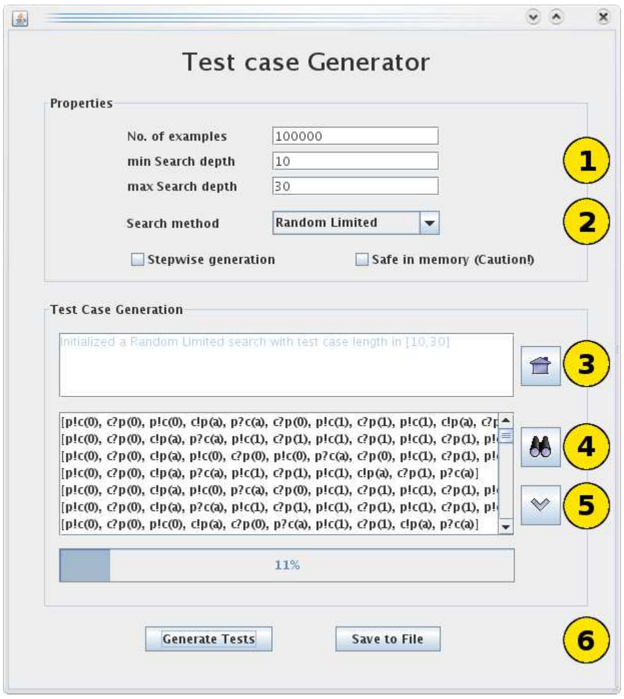


Figure 9.17: The model-based testing approach using `Smyle` and `Style`



No. of examples 100000
min Search depth 10
max Search depth 30
Search method Random Limited
☐ Stepwise generation ☐ Safe in memory (Caution!)

```
Initialized a Random Limited search with test case length in [10,30]
```

```
[p!c(0), c?p(0), p!c(0), c!p(a), p?c(a), c?p(0), p!c(1), c?p(1), p!c(1), c!p(a), c?p
[p!c(0), c?p(0), c!p(a), p?c(a), p!c(1), c?p(1), p!c(1), c?p(1), p!c(1), c?p(1), p!
[p!c(0), c?p(0), c!p(a), p!c(0), c?p(0), p!c(0), p?c(a), c?p(0), p!c(1), c?p(1), p!
[p!c(0), c?p(0), c!p(a), p?c(a), p!c(1), c?p(1), p!c(1), c!p(a), c?p(1), p?c(a)]
[p!c(0), c?p(0), c!p(a), p!c(0), p?c(a), c?p(0), p!c(1), c?p(1), p!c(1), c?p(1), p!
[p!c(0), c?p(0), c!p(a), p?c(a), p!c(1), c?p(1), p!c(1), c?p(1), p!c(1), c?p(1), p!
[p!c(0), c?p(0), p!c(0), c!p(a), c?p(0), p?c(a), p!c(1), c?p(1), c!p(a), p?c(a)]
```

11%

Figure 9.18: Test case generation using `Style`

mentations, and can generate test cases of a priori fixed length specified by an interval `[minimalLength,maximalLength]`. In Figure 9.18, for example, we are deriving a test suite of 100,000 test cases of length between 10 and 30 events (cf. Figure 9.18 (1)) using a random search method (cf. Figure 9.18 (2)). Other features of the test case generator GUI are: component (3) for initializing the test case generation, component `(4)` for depicting single test cases, component `(5)` for stepwise generation of test cases, and the buttons from item `(6)` to generate the test suite specified in components (1) and (2), and saving this test suite to disk. Figure 9.19 shows four sample test cases that were generated using `Style` in the previously described setting. More involved algorithms employed for test case generation still have to be implemented (e.g., [Vas73, Cho78, FvBK^{+}91]).

For now, `Style` is only capable of deriving non-parameterized test cases, but we are planning to extend it to parameterized test case generation, where parameterized traces and MSCs are generated from the abstract model, yielding a symbolic approach to automated test case generation. Instead of instantiating the test cases with concrete values, we let the tests execute on a symbolic level and check on-the-fly whether the execution satisfies the test. Another possibility when given a parameterized set of tests is to generate new sets of concrete test cases by instantiating the abstract test cases on basis of the infor-

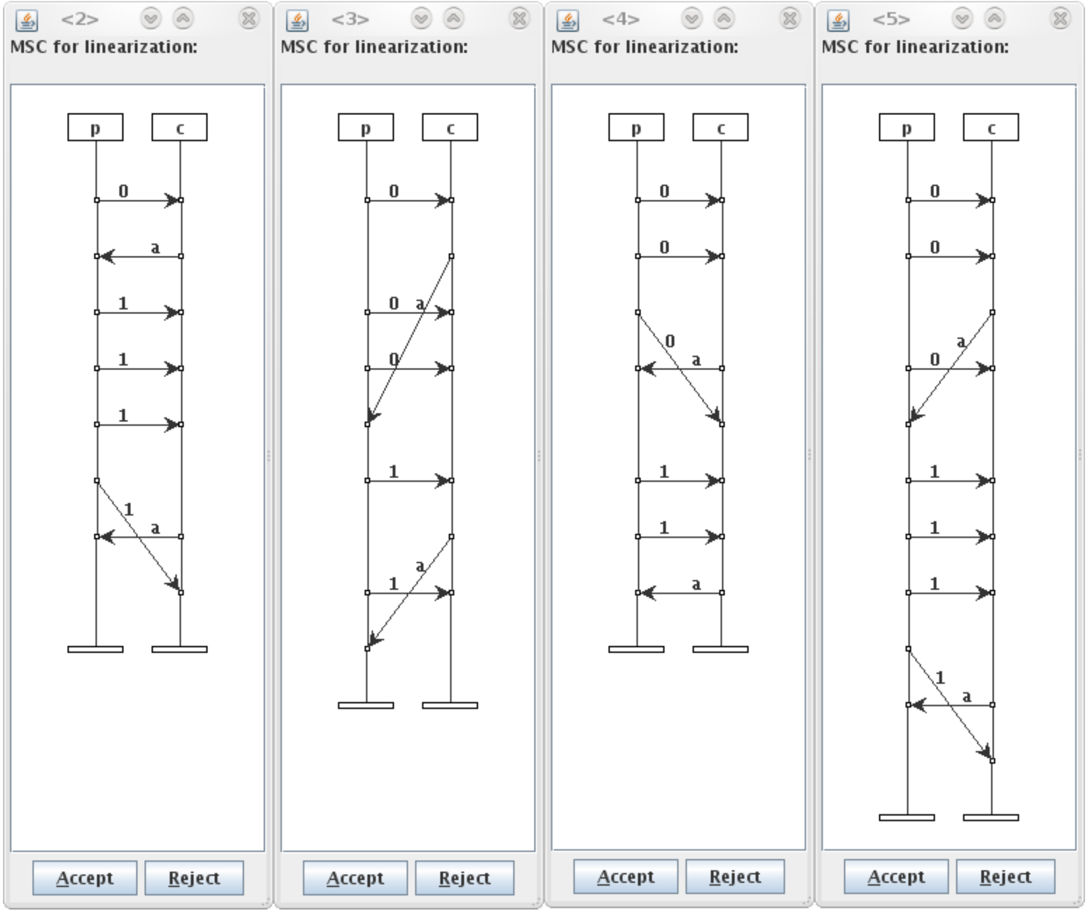

<2>
MSC for linearization:
p
c
0
a
1
1
1
1
a
Accept
Reject
<3>
MSC for linearization:
p
c
0
0
a
0
1
a
1
Accept
Reject
<4>
MSC for linearization:
p
c
0
0
0
a
1
1
a
Accept
Reject
<5>
MSC for linearization:
p
c
0
0
a
0
1
1
1
1
a
Accept
Reject

Figure 9.19: Four positive test cases for the ABP (cf. Sections 7.4 and 9.2) generated by `Style` using the test case setup specified in Figure 9.18 (1), (2)

mation about data types stored or provided by the user for the parameters. The future version of `Style` will thus provide a powerful means to generate large size, parameterized test suites for model-based testing.

10 Conclusions and Outlook

This thesis generalized Angluin's learning algorithm L^* in several ways. On the one hand, a new active online algorithm—called NL^*—for learning NFA in the spirit of the L^* algorithm was introduced. To the best of our knowledge it is the first and only of its kind. Though its theoretical time complexity is slightly worse than that obtained for L^*, practical experiments show substantially better outcomes than for L^*. This suggests that a better worst-case complexity than that of L^* might be derivable.

A further improvement of table-based learning algorithms in general was introduced under the name *learning of congruence-closed languages* and employed in the setting of distributed system synthesis as *partial order learning*. Several case studies showed the advantage of this approach where memory savings of up to 70% and membership query reductions of more than 75% could be achieved. The successful application of our congruence-closed language learning approach depends on domain-specific properties. As such congruence-closed language learning always has to be tailored to the target domain, and will be applicable to many (but not necessary all) domains.

Moreover, we extended Angluin's learning approach for synthesizing DFA from regular languages to a procedure that infers CFMs from sets of positive and negative scenarios given as basic MSCs. Note that this technique, in contrast to many other existing approaches, is performed in an asynchronous setting and yields an exact approach—the resulting CFM precisely accepts the positive MSCs and rejects the negative ones—and is applicable to various classes of CFMs such as different types of universally bounded CFMs and to the class of existentially bounded CFMs. Our learning setting is also applicable to other classes like, e.g., the *causal closure* as defined by Adsul et al. [AMKN05]. However, it remains open whether the class of weak CFMs is learnable.

We have shown the feasibility of our approach by reporting on some experiments that we carried out employing our tool `Smyle`[1]. By exploiting the properties of partial orders (like MSCs) a significant reduction of the memory consumption could be achieved. Alternative improvements covered in this thesis were, e.g., the reduction of the number of user queries (i.e., of the queries that are presented in our tool `Smyle`) by using the logic PDL [BKM07] to specify a priori undesired partial behavior in terms of so-called *patterns*, e.g., in case of the ABP "no bit change without prior acknowledgment". First results towards using PDL to this purpose have been presented in Chapter 7 and recently reported in [BKKL09, BKKL10]. Chapter 7 also described how to embed `Smyle` into an incremental software engineering process called `SMA`.

Furthermore, we developed a learning library called `libalf` that implements most of the learning algorithms mentioned in this thesis as well as their extensions (L^*_{col}, partial-order learning, etc.). We also integrated this library into `Smyle` and used it to infer the protocols mentioned in Section 9.2. Initial results obtained with this learning framework are promising, and several research groups, e.g., from University of Oxford, TU Munich, and RWTH Aachen University, already evinced interest in participating or using `libalf`.

---

[1] `Smyle` is freely available for exploration at `http://www.smyle-tool.org/`.

Future Work

Though the results of this thesis significantly extend the current state of the art of grammatical inference, there is still space for further improvement. A feature which is currently not implemented in our tool `Smyle` but could be of great help for designing protocols is to support co-regions in basic MSCs (because, yet representing basic MSCs, they can assemble several basic MSCs in one picture). A possible direction for future work is to investigate further classes of learnable CFMs, like *causal closure* MSC languages [AMKN05], which could enrich our learning tool `Smyle`.

Concerning the learning of compact representations of (ω-)regular languages, we already spotted two directions in which learning of nondeterministic automata could be extended.

The first enhancement would be to consider the class (or a subclass) of (residual) alternating finite-state automata (RAFA) and to develop an active online algorithm to derive such target automata. As, for a given regular language, alternating automata can be doubly-exponentially more succinct than their corresponding minimal DFA, this class could provide another, even more compact representation of regular languages than minimal DFA or canonical RFSA.

A further field of interest are ω-regular languages because they can describe behavior of reactive (i.e., possibly infinitely-long running) systems. There are already algorithms for learning (subclasses) of ω-regular languages. One approach is able to infer weak deterministic Büchi automata [MP95]. It might be worth considering the notion of residual weak Büchi automata (RWBA) in order to derive nondeterministic Büchi automata which—as in the finite case—might be exponentially more succinct than the equivalent deterministic automata. Another recent approach [FCC$^+$08] in this field is capable of deriving nondeterministic Büchi automata by learning projections of ω-regular languages to regular languages. They use a slight extension of the L^* algorithm and transform the inferred DFA into nondeterministic Büchi automata recognizing the correct ω-regular languages. Thereby, they are able to cover the whole class of ω-regular languages. But even in this setting, the use of residual languages and the application of NL^* might be sensible in order to infer exponentially more succinct intermediate automata (instead of DFA using L^*). This way, it might be possible to infer residual (weak) Büchi automata, and again obtain an exponential gain compared to their deterministic counterpart.

Moreover, it would be interesting to employ our algorithm NL^* for formal verification tasks. As the finite-state automata derived by NL^* may be exponentially more succinct than the corresponding minimal DFA, NL^* might be a better choice for verification applications. A concrete idea is to check whether our new learning algorithm performs well in the area of *regular model checking* [BJNT00, AJNS04, HV05]. The latter paper presents a technique that employs a passive offline learning algorithm by Trakhtenbrot and Barzdin [TB73] to perform regular model checking.

A Auxiliary Calculations for Chapter 2

Example A.0.1. We now give some auxiliary calculations for Example 2.3.9 from page 14 given the regular language $L = L(((a^*|b^*)a\Sigma)^*))$: to simplify the presentation, we write rL or Lr instead of $L(r) \cdot L$ or $L \cdot L(r)$, respectively (for a regular expression r and a regular language L). Moreover, for a regular expression $r \in \mathfrak{R}$, $r^+ := r \cdot r^*$. With this definition we get:

- $\varepsilon^{-1}L = L = L_{p_0}$ (by definition of residual languages and deterministic automata),
- $(aa)^{-1}L = (a^*a\Sigma|\Sigma|\varepsilon)L = (a^*|a^+b|b)L = (a^*|(a^+|\varepsilon)b)L = (a^*|a^*b)L = L_{p_2}$,
- $a^{-1}L = (a^*a\Sigma|a\Sigma|\Sigma)L = (a^*\Sigma)L = (a^+|a^*b)L = a^+L|a^+bL|bL =$ $(a^+|a^+b)L|a^+bL|bL = a^+((a^*|a^*b)L)|a^+bL|bL = a^+L_{p_2}|aa^*bL|bL = L_{p_1}$,
- $b^{-1}L = (b^*a\Sigma|a\Sigma)L = (b^*a\Sigma)L = L_{p_3}$, and
- $(ba)^{-1}L = \Sigma L = L_{p_4}$.

◇

Example A.0.2. In the following we will calculate the languages of the states for Example 2.3.12 on page 15. First, we compile the following equation system for the NFA from Figure 2.3(b) on page 15:

- $L_{s_0} = \Sigma L_{s_1}|aL_{s_2}|\varepsilon$,
- $L_{s_1} = b^*aL_{s_2}$, and
- $L_{s_2} = \Sigma L_{s_0}$.

Using backward propagation yields:

- $L_{s_0} = \Sigma L_{s_1}|aL_{s_2}|\varepsilon = \Sigma b^*a\Sigma L_{s_0}|a\Sigma L_{s_0}|\varepsilon$,
- $L_{s_1} = b^*aL_{s_2} = b^*a\Sigma L_{s_0}$, and
- $L_{s_2} = \Sigma L_{s_0}$.

Applying the result from [Ard60] and[Ard61] for solving regular equations, which says that every regular equation of the form $X = r_1 \mid r_2X$ (where, $r_1, r_2 \in \mathfrak{R}$ and $\varepsilon \notin L(r_2)$) admits a minimal solution: $X = r_2^* \cdot r_1$, we finally obtain:

- $L_{s_0} = L(((\Sigma b^* \mid \varepsilon)\, a\Sigma)^*) = L\,(((ab^* \mid b^*)\, a\Sigma)^*) = L\,(((a^* \mid b^*)\, a\Sigma)^*) = L_{p_0} = L$ (where p_0 is the initial state of the minimal DFA $\mathcal{A}$ from Figure 2.1(a) on page 12), and hence:
- $L_{s_1} = b^*a\Sigma L = L_{p_3}$, and
- $L_{s_2} = \Sigma L = L_{p_4}$,

where L_{p_0}, L_{p_3} and L_{p_4} are residual languages from L and describe the languages accepted by states p_0, p_3 and p_4 from Figure 2.1(a).

◇

B Insights on RFSA Learning (Interesting Examples)

In the following, we will describe interesting examples showing certain characteristics of the NL* algorithm presented in Chapter 4.

B.1 An example where NL* needs more membership queries than L*

In this section, we see an example where NL* needs more membership queries than its deterministic version L*. Moreover, the resulting automata have got the same number of states.

Let a parameterized minimal DFA over $\Sigma = \{a, b\}$ be $\mathcal{A}_n^* = (Q, \{q_0\}, \delta, F)$ (for $n > 1$), let $\mathcal{R}_n$ be the corresponding canonical RFSA, and let $L_n = L(\mathcal{A}_n^*) = L(\mathcal{R}_n)$. Hereby, $\mathcal{A}_n^*$ is given by:

- $Q = \{q_i \,|\, 0 \leq i \leq n\}$,
- $\delta(q_i, a) = \delta(q_i, b) = q_{i+1}$ for $0 \leq i \leq n-2$, $\delta(q_{n-1}, a) = q_n$, $\delta(q_{n-1}, b) = q_0$, $\delta(q_n, a) = \delta(q_n, b) = q_n$, and
- $F = Q \setminus \{q_n\}$.

Figure B.1 shows the minimal DFA $\mathcal{A}_4^*$ and Figure B.2 the corresponding canonical RFSA $\mathcal{R}_4$.

Lemma B.1.1 (Learning RFSA might need more membership queries than learning DFA)**.** *In comparison to learning the minimal DFA $\mathcal{A}_n^*$ using L*, learning the canonical RFSA $\mathcal{R}_n$ requires $n-1$ additional resolutions of inconsistencies (but no additional equivalence queries).*

Proof: The proof can be done via induction on the number of states and is easily traceable following the example below. Note that NL* needs indeed $(2n+1) \cdot (n-2)$ more membership queries than L*. ☐

Learning the DFA from Figure B.1 and its canonical RFSA (Figure B.2) is depicted in Tables B.1 and B.2, respectively.

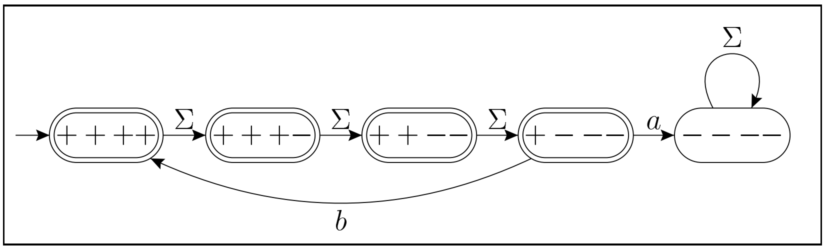


Figure B.1: The minimal DFA $\mathcal{A}_4^*$ over $\Sigma = \{a, b\}$ for which NL* needs more membership queries than L*

$\mathcal{T}_0$	ε
ε	+
a	+
b	+

$\overset{1)}{\Rightarrow}$

$\mathcal{T}_1$	ε
ε	+
a	+
aa	+
aaa	+
$aaaa$	−
b	+
ab	+
aab	+
$aaab$	+
$aaaaa$	−
$aaaab$	−

$\overset{2)}{\Rightarrow}$

$\mathcal{T}_2$	ε	a
ε	+	+
a	+	+
aa	+	+
aaa	+	−
$aaaa$	−	−
b	+	+
ab	+	+
aab	+	−
$aaab$	+	+
$aaaaa$	−	−
$aaaab$	−	−

$\overset{3)}{\Rightarrow}$

$\mathcal{T}_3$	ε	a	aa
ε	+	+	+
a	+	+	+
aa	+	+	−
aaa	+	−	−
$aaaa$	−	−	−
b	+	+	+
ab	+	+	−
aab	+	−	−
$aaab$	+	+	+
$aaaaa$	−	−	−
$aaaab$	−	−	−

$\overset{4)}{\Rightarrow}$

$\mathcal{T}_4$	ε	a	aa	aaa
ε	+	+	+	+
a	+	+	+	−
aa	+	+	−	−
aaa	+	−	−	−
$aaaa$	−	−	−	−
b	+	+	+	−
ab	+	+	−	−
aab	+	−	−	−
$aaab$	+	+	+	+
$aaaaa$	−	−	−	−
$aaaab$	−	−	−	−

1) $\mathcal{T}_0$ is closed and consistent, but a counterexample can be obtained because $aaaa \in L(\mathcal{A}_{\mathcal{T}_1})$ and $aaaa \notin L$. Hence, add $\mathit{pref}(aaaa)$ to U.

2) $\mathcal{T}_1$ is not consistent as $T(aa) = T(aaa)$ but $T(aaa) \neq T(aaaa)$. Hence, add a to V.

3) $\mathcal{T}_2$ is not consistent as $T(a) = T(aa)$ but $T(aa) \neq T(aaa)$. Hence, add aa to V.

4) $\mathcal{T}_3$ is not consistent as $T(\varepsilon) = T(a)$ but $T(a) \neq T(aa)$. Hence, add aaa to V.

5) $\mathcal{T}_4$ is then closed and consistent, and the minimal DFA $\mathcal{A}_4^*$ from Figure B.1 can be derived.

Table B.1: Learning $\mathcal{A}_5^*$ with L*

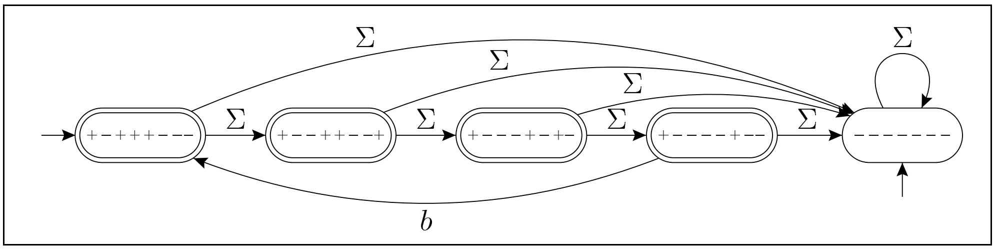


Figure B.2: RFSA $\mathcal{R}_4$ recognizing the language of the minimal DFA $\mathcal{A}_4^*$ from Figure B.1

	$\mathcal{T}_0$	ε
$*$	ε	+
$*$	b	+
$*$	a	+

$\Rightarrow^{1.}_{ce}$

	$\mathcal{T}_1$	ε	$aaaa$	aaa	aa	a
$*$	ε	+	−	+	+	+
$*$	b	+	−	−	+	+
$*$	a	+	−	−	+	+

$\Rightarrow^{2.}_{ncl}$

	$\mathcal{T}_2$	ε	$aaaa$	aaa	aa	a
$*$	ε	+	−	+	+	+
$*$	b	+	−	−	+	+
$*$	a	+	−	−	+	+
$*$	bb	+	−	−	−	+
$*$	ba	+	−	−	−	+

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	$aaaa$	aaa	aa	a
$*$	ε	+	−	+	+	+
$*$	b	+	−	−	+	+
$*$	bb	+	−	−	−	+
$*$	a	+	−	−	+	+
$*$	ba	+	−	−	−	+
$*$	bbb	+	−	−	−	−
$*$	bba	+	−	−	−	−

$\Rightarrow^{4.}_{ncl}$

	$\mathcal{T}_4$	ε	$aaaa$	aaa	aa	a
$*$	ε	+	−	+	+	+
$*$	b	+	−	−	+	+
$*$	bb	+	−	−	−	+
$*$	bbb	+	−	−	−	−
$*$	a	+	−	−	+	+
$*$	ba	+	−	−	−	+
$*$	bba	+	−	−	−	−
$*$	$bbbb$	+	−	+	+	+
$*$	$bbba$	−	−	−	−	−

$\Rightarrow^{5.}_{ncl}$

	$\mathcal{T}_5$	ε	$aaaa$	aaa	aa	a
$*$	ε	+	−	+	+	+
$*$	b	+	−	−	+	+
$*$	bb	+	−	−	−	+
$*$	bbb	+	−	−	−	−
$*$	$bbba$	−	−	−	−	−
$*$	a	+	−	−	+	+
$*$	ba	+	−	−	−	+
$*$	bba	+	−	−	−	−
$*$	$bbbb$	+	−	+	+	+
$*$	$bbbab$	−	−	−	−	−
$*$	$bbbaa$	−	−	−	−	−

$\Rightarrow^{6.}_{ncs}$

	$\mathcal{T}_6$	ε	$aaaa$	aaa	aa	a	$baaa$
$*$	ε	+	−	+	+	+	−
$*$	b	+	−	−	+	+	−
$*$	bb	+	−	−	−	+	−
$*$	bbb	+	−	−	−	−	+
$*$	$bbba$	−	−	−	−	−	−
$*$	a	+	−	−	+	+	−
$*$	ba	+	−	−	−	+	−
$*$	bba	+	−	−	−	−	+
$*$	$bbbb$	+	−	+	+	+	−
$*$	$bbbab$	−	−	−	−	−	−
$*$	$bbbaa$	−	−	−	−	−	−

$\Rightarrow^{7.}_{ncs}$

	$\mathcal{T}_7$	ε	$aaaa$	aaa	aa	a	$baaa$	$bbaaa$
$*$	ε	+	−	+	+	+	−	−
$*$	b	+	−	−	+	+	−	−
$*$	bb	+	−	−	−	+	−	+
$*$	bbb	+	−	−	−	−	+	−
$*$	$bbba$	−	−	−	−	−	−	−
$*$	a	+	−	−	+	+	−	−
$*$	ba	+	−	−	−	+	−	+
$*$	bba	+	−	−	−	−	+	−
$*$	$bbbb$	+	−	+	+	+	−	−
$*$	$bbbab$	−	−	−	−	−	−	−
$*$	$bbbaa$	−	−	−	−	−	−	−

$\Rightarrow^{8.}_{ncs}$

	$\mathcal{T}_8$	ε	$aaaa$	aaa	aa	a	$baaa$	$bbaaa$	$bbbaaa$
$*$	ε	+	−	+	+	+	−	−	−
$*$	b	+	−	−	+	+	−	−	+
$*$	bb	+	−	−	−	+	−	+	−
$*$	bbb	+	−	−	−	−	+	−	−
$*$	$bbba$	−	−	−	−	−	−	−	−
$*$	a	+	−	−	+	+	−	−	+
$*$	ba	+	−	−	−	+	−	+	−
$*$	bba	+	−	−	−	−	+	−	−
$*$	$bbbb$	+	−	+	+	+	−	−	−
$*$	$bbbab$	−	−	−	−	−	−	−	−
$*$	$bbbaa$	−	−	−	−	−	−	−	−

1) Table $\mathcal{T}_0$ is RFSA-closed and RFSA-consistent, but a counterexample can be found: *aaaa* is accepted by the hypothesis but is not in the language we want to infer.

2) Table $\mathcal{T}_1$ violates the RFSA-closedness property. Hence, we try to make $\mathcal{T}_1$ RFSA-closed by moving row b to U.

3)−5) We get three more RFSA-closedness violations and resolve them by moving rows *bb*, *bbb* and *bbba* to the upper part of the table.

6) Table $\mathcal{T}_5$ violates the RFSA-consistency property because $bbb \sqsubseteq bb$ but $bbbb \not\sqsubseteq bbb$. We try to obtain RFSA-consistency by adding suffixes *baaa* to V.

7)−8) But still, two more inconsistencies occur. Hence we add suffix *bbaaa* and *bbbaaa* to V.

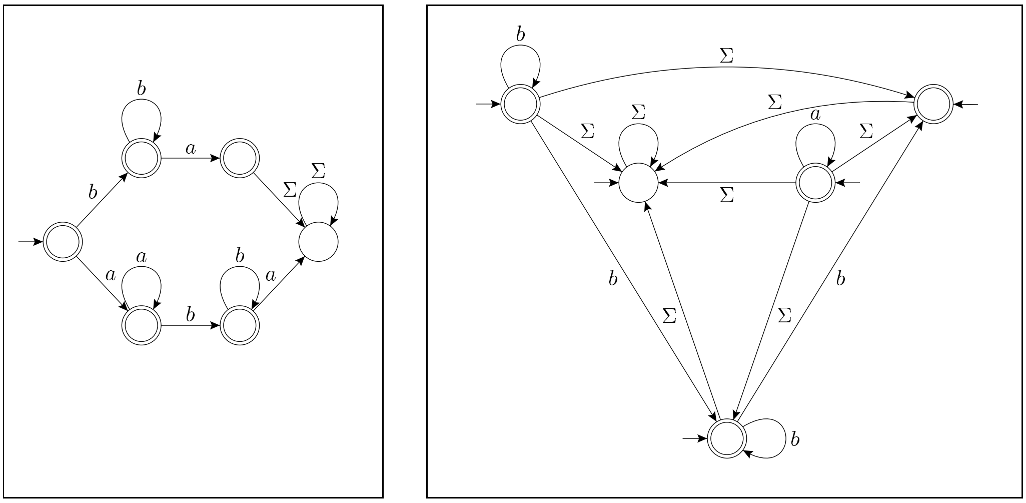
9) Table $\mathcal{T}_8$ is RFSA-closed and RFSA-consistent, and the final model $\mathcal{R}_4$ can be calculated (cf. Figure B.2).

Table B.2: Learning $\mathcal{R}_4$ with NL*

B.2 An example where NL* needs more equivalence queries than L*

After describing an example where NL* needed more membership queries than L*, we now turn to an example that shows that in some cases even the number of equivalence queries can be slightly larger than for L*.

Let L be the regular language accepted by the minimal DFA $\mathcal{A}_{\mathcal{T}_6}$ from Figure B.3(a). Even though the equivalent canonical RFSA $\mathcal{R}_{\mathcal{T}_8}$ has less states than $\mathcal{A}_{\mathcal{T}_6}$, to infer $\mathcal{R}_{\mathcal{T}_8}$ needs more equivalence queries (using NL*) than to infer $\mathcal{A}_{\mathcal{T}_6}$ (using L*).



(a) Minimal DFA $\mathcal{A}_{\mathcal{T}_6}$

(b) Canonical RFSA $\mathcal{R}_{\mathcal{T}_8}$

Figure B.3: Acceptors for regular language L

$\mathcal{T}_0$	ε
ε	+
b	+
a	+

$\Rightarrow^{1.}_{ce}$

$\mathcal{T}_1$	ε
ε	+
a	+
aba	−
ab	+
b	+
aa	+
$abab$	−
$abaa$	−
abb	+

$\Rightarrow^{2.}_{ncs}$

$\mathcal{T}_2$	ε	a
ε	+	+
a	+	+
aba	−	−
ab	+	−
b	+	+
aa	+	+
$abab$	−	−
$abaa$	−	−
abb	+	−

$\Rightarrow^{3.}_{ncs}$

$\mathcal{T}_3$	ε	a	ba
ε	+	+	+
a	+	+	−
aba	−	−	−
ab	+	−	−
b	+	+	+
aa	+	+	−
$abab$	−	−	−
$abaa$	−	−	−
abb	+	−	−

$\Rightarrow^{4.}_{ce}$

$\mathcal{T}_4$	ε	a	ba
ε	+	+	+
b	+	+	+
a	+	+	−
aba	−	−	−
ab	+	−	−
baa	−	−	−
ba	+	−	−
aa	+	+	−
$abab$	−	−	−
$abaa$	−	−	−
abb	+	−	−
bb	+	+	+
$baab$	−	−	−
$baaa$	−	−	−
bab	−	−	−

$\Rightarrow^{5.}_{ncs}$

$\mathcal{T}_5$	ε	a	ba	aa
ε	+	+	+	+
b	+	+	+	−
a	+	+	−	+
aba	−	−	−	−
ab	+	−	−	−
baa	−	−	−	−
ba	+	−	−	−
aa	+	+	−	+
$abab$	−	−	−	−
$abaa$	−	−	−	−
abb	+	−	−	−
bb	+	+	+	−
$baab$	−	−	−	−
$baaa$	−	−	−	−
bab	−	−	−	−

$\Rightarrow^{6.}_{ncs}$

$\mathcal{T}_6$	ε	a	ba	aa	b
ε	+	+	+	+	+
b	+	+	+	−	+
a	+	+	−	+	+
aba	−	−	−	−	−
ab	+	−	−	−	+
baa	−	−	−	−	−
ba	+	−	−	−	−
aa	+	+	−	+	+
$abab$	−	−	−	−	−
$abaa$	−	−	−	−	−
abb	+	−	−	−	+
bb	+	+	+	−	+
$baab$	−	−	−	−	−
$baaa$	−	−	−	−	−
bab	−	−	−	−	−

1) Found counterexample aba for current model $\mathcal{A}_{\mathcal{T}_0}$.
2) $\mathcal{T}_1$ is not consistent. Trying to obtain consistency by adding suffix a to V.
3) $\mathcal{T}_2$ is not consistent. Trying to obtain consistency by adding suffix ba to V.
4) Found counterexample baa for current model $\mathcal{A}_{\mathcal{T}_3}$.
5) $\mathcal{T}_4$ is not consistent. Trying to obtain consistency by adding suffix aa to V.
6) $\mathcal{T}_5$ is not consistent. Trying to obtain consistency by adding suffix b to V.
7) Table $\mathcal{T}_6$ is closed and consistent, and the final model $\mathcal{A}_{\mathcal{T}_6}$ is calculated (cf. Figure B.3(a)).

Table B.3: An example of an L* run that needs less equivalence queries than NL*

	$\mathcal{T}_0$	ε
$*$	ε	+
$*$	b	+
$*$	a	+

$\Rightarrow^{1.}_{ce}$

	$\mathcal{T}_1$	ε	aba	ba	a
$*$	ε	+	−	+	+
$*$	b	+	−	+	+
$*$	a	+	−	−	+

$\Rightarrow^{2.}_{ncl}$

	$\mathcal{T}_2$	ε	aba	ba	a
$*$	ε	+	−	+	+
$*$	a	+	−	−	+
$*$	b	+	−	+	+
$*$	ab	+	−	−	−
$*$	aa	+	−	−	+

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	aba	ba	a
$*$	ε	+	−	+	+
$*$	a	+	−	−	+
$*$	ab	+	−	−	−
$*$	b	+	−	+	+
$*$	aa	+	−	−	+
$*$	abb	+	−	−	−
$*$	aba	−	−	−	−

$\Rightarrow^{4.}_{ncl}$

	$\mathcal{T}_4$	ε	aba	ba	a
$*$	ε	+	−	+	+
$*$	a	+	−	−	+
$*$	ab	+	−	−	−
$*$	aba	−	−	−	−
$*$	b	+	−	+	+
$*$	aa	+	−	−	+
$*$	abb	+	−	−	−
$*$	$abab$	−	−	−	−
$*$	$abaa$	−	−	−	−

$\Rightarrow^{5.}_{ce}$

	$\mathcal{T}_5$	ε	aba	ba	a	baa	aa
	ε	+	−	+	+	−	+
$*$	a	+	−	−	+	−	+
$*$	ab	+	−	−	−	−	−
$*$	aba	−	−	−	−	−	−
$*$	b	+	−	+	+	−	−
$*$	aa	+	−	−	+	−	+
$*$	abb	+	−	−	−	−	−
$*$	$abab$	−	−	−	−	−	−
$*$	$abaa$	−	−	−	−	−	−

$\Rightarrow^{6.}_{ncl}$

	$\mathcal{T}_6$	ε	aba	ba	a	baa	aa
	ε	+	−	+	+	−	+
$*$	b	+	−	+	+	−	−
$*$	a	+	−	−	+	−	+
$*$	ab	+	−	−	−	−	−
$*$	aba	−	−	−	−	−	−
$*$	aa	+	−	−	+	−	+
$*$	abb	+	−	−	−	−	−
$*$	$abab$	−	−	−	−	−	−
$*$	$abaa$	−	−	−	−	−	−
$*$	bb	+	−	+	+	−	−
$*$	ba	+	−	−	−	−	−

$\Rightarrow^{7.}_{ce}$

	$\mathcal{T}_7$	ε	aba	ba	a	baa	aa	bab	ab	b
	ε	+	−	+	+	−	+	−	+	+
$*$	b	+	−	+	+	−	−	−	−	+
$*$	a	+	−	−	+	−	+	−	+	+
$*$	ab	+	−	−	−	−	−	−	−	+
$*$	aba	−	−	−	−	−	−	−	−	−
$*$	aa	+	−	−	+	−	+	−	+	+
$*$	abb	+	−	−	−	−	−	−	−	+
$*$	$abab$	−	−	−	−	−	−	−	−	−
$*$	$abaa$	−	−	−	−	−	−	−	−	−
$*$	bb	+	−	+	+	−	−	−	−	+
$*$	ba	+	−	−	−	−	−	−	−	−

$\Rightarrow^{8.}_{ncl}$

	$\mathcal{T}_8$	ε	aba	ba	a	baa	aa	bab	ab	b
	ε	+	−	+	+	−	+	−	+	+
$*$	b	+	−	+	+	−	−	−	−	+
$*$	a	+	−	−	+	−	+	−	+	+
$*$	ab	+	−	−	−	−	−	−	−	+
$*$	aba	−	−	−	−	−	−	−	−	−
$*$	ba	+	−	−	−	−	−	−	−	−
$*$	aa	+	−	−	+	−	+	−	+	+
$*$	abb	+	−	−	−	−	−	−	−	+
$*$	$abab$	−	−	−	−	−	−	−	−	−
$*$	$abaa$	−	−	−	−	−	−	−	−	−
$*$	bb	+	−	+	+	−	−	−	−	+
$*$	bab	−	−	−	−	−	−	−	−	−
$*$	baa	−	−	−	−	−	−	−	−	−

1) Found counterexample aba for current model $\mathcal{R}_{\mathcal{T}_0}$.
2) $\mathcal{T}_1$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row a to U.
3) $\mathcal{T}_2$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row ab to U.
4) $\mathcal{T}_3$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row aba to U.
5) Found counterexample baa for current model $\mathcal{R}_{\mathcal{T}_4}$.
6) $\mathcal{T}_5$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row b to U.
7) Found counterexample bab for current model $\mathcal{R}_{\mathcal{T}_6}$.
8) $\mathcal{T}_7$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row ba to U.
9) Table $\mathcal{T}_8$ is RFSA-closed and RFSA-consistent, and the final model $\mathcal{R}_{\mathcal{T}_8}$ is calculated (cf. Figure B.3(b)).

Table B.4: An example of an NL* run that needs more equivalence queries than L*

B.3 An example where the intermediate hypothesis is not an RFSA

As mentioned in Chapter 4, the NL* algorithm for inferring canonical RFSA does not necessarily infer (canonical) RFSA as intermediate hypotheses. A nice example is given in Figure B.5. Depicted are the intermediate hypotheses automata $\mathcal{R}_{\mathcal{T}_1}$ and $\mathcal{R}_{\mathcal{T}_4}$ (for tables $\mathcal{T}_1$ and $\mathcal{T}_4$, respectively), and the final hypothesis $\mathcal{R}_{\mathcal{T}_6}$ (cf. Figure B.5(c)). Note that $\mathcal{R}_{\mathcal{T}_4}$ (cf. Figure B.5(b)) is not an RFSA, because the state at the bottom accepts bbb^*, which is not a residual of $L(\mathcal{R}_{\mathcal{T}_4}) = bb^*$. The final hypothesis $\mathcal{R}_{\mathcal{T}_6}$, however, is a canonical RFSA.

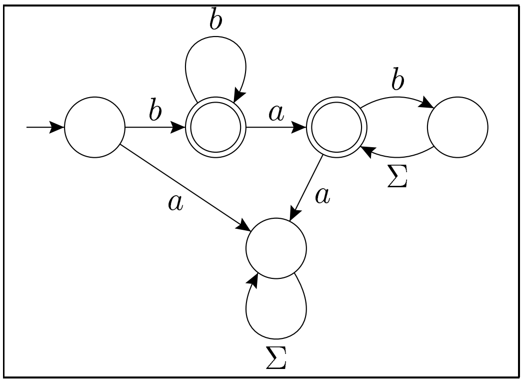
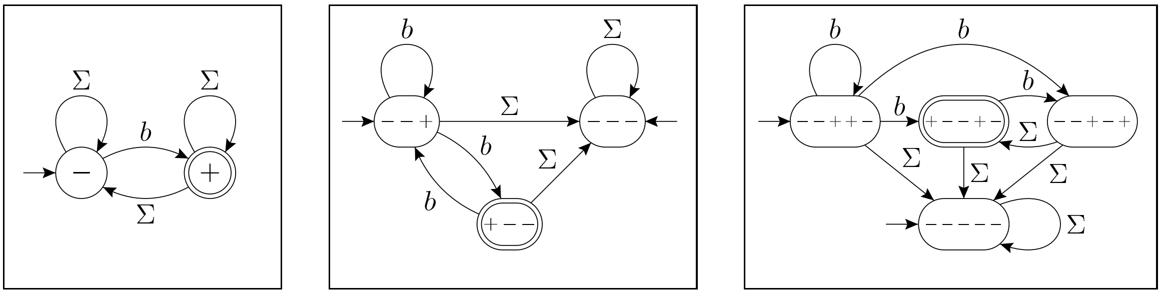


Figure B.4: Minimal DFA recognizing the language to infer



(a) Hypothesis $\mathcal{R}_{\mathcal{T}_1}$ (b) Hypothesis $\mathcal{R}_{\mathcal{T}_4}$ (c) canonical RFSA $\mathcal{R}_{\mathcal{T}_6}$

Figure B.5: All hypotheses for inferring the canonical RFSA for minimal DFA from Figure B.4

	$\mathcal{T}_0$	ε
$*$	ε	$-$
$*$	b	$+$
$*$	a	$-$

$\Rightarrow^{1.}_{ncl}$

	$\mathcal{T}_1$	ε
$*$	ε	$-$
$*$	b	$+$
$*$	a	$-$
$*$	bb	$+$
$*$	ba	$+$

$\Rightarrow^{2.}_{ce}$

	$\mathcal{T}_2$	ε	ab	b
$*$	ε	$-$	$-$	$+$
	b	$+$	$-$	$+$
$*$	a	$-$	$-$	$-$
	bb	$+$	$-$	$+$
$*$	ba	$+$	$-$	$-$

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	ab	b
$*$	ε	$-$	$-$	$+$
	b	$+$	$-$	$+$
$*$	a	$-$	$-$	$-$
	bb	$+$	$-$	$+$
$*$	ba	$+$	$-$	$-$
$*$	ab	$-$	$-$	$-$
$*$	aa	$-$	$-$	$-$

$\Rightarrow^{4.}_{ncl}$

	$\mathcal{T}_4$	ε	ab	b
$*$	ε	$-$	$-$	$+$
	b	$+$	$-$	$+$
$*$	a	$-$	$-$	$-$
$*$	ba	$+$	$-$	$-$
	bb	$+$	$-$	$+$
$*$	ab	$-$	$-$	$-$
$*$	aa	$-$	$-$	$-$
$*$	bab	$-$	$-$	$+$
$*$	baa	$-$	$-$	$-$

$\Rightarrow^{5.}_{ce}$

	$\mathcal{T}_5$	ε	ab	b	ba	a
$*$	ε	$-$	$-$	$+$	$+$	$-$
	b	$+$	$-$	$+$	$+$	$+$
$*$	a	$-$	$-$	$-$	$-$	$-$
$*$	ba	$+$	$-$	$-$	$+$	$-$
	bb	$+$	$-$	$+$	$+$	$+$
$*$	ab	$-$	$-$	$-$	$-$	$-$
$*$	aa	$-$	$-$	$-$	$-$	$-$
$*$	bab	$-$	$-$	$+$	$-$	$+$
$*$	baa	$-$	$-$	$-$	$-$	$-$

$\Rightarrow^{6.}_{ncl}$

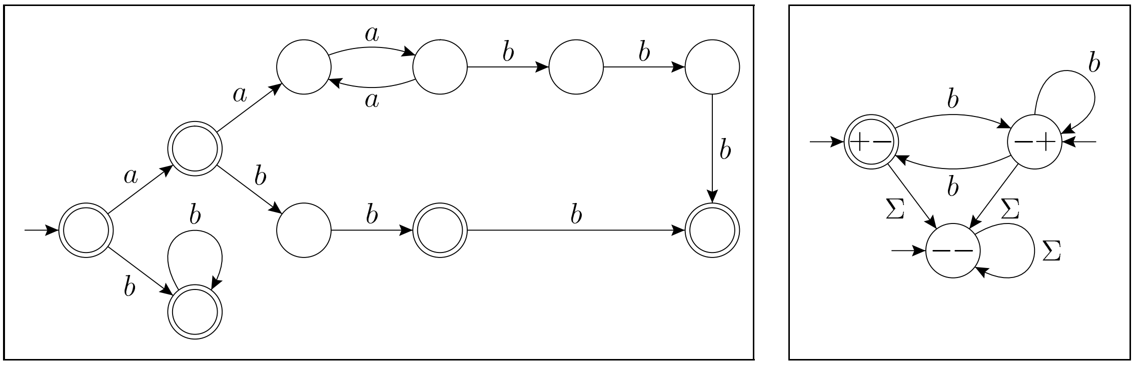
	$\mathcal{T}_6$	ε	ab	b	ba	a
$*$	ε	$-$	$-$	$+$	$+$	$-$
	b	$+$	$-$	$+$	$+$	$+$
$*$	a	$-$	$-$	$-$	$-$	$-$
$*$	ba	$+$	$-$	$-$	$+$	$-$
$*$	bab	$-$	$-$	$+$	$-$	$+$
	bb	$+$	$-$	$+$	$+$	$+$
$*$	ab	$-$	$-$	$-$	$-$	$-$
$*$	aa	$-$	$-$	$-$	$-$	$-$
$*$	baa	$-$	$-$	$-$	$-$	$-$
$*$	$babb$	$+$	$-$	$-$	$+$	$-$
$*$	$baba$	$+$	$-$	$-$	$+$	$-$

1) $\mathcal{T}_0$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row b to U.
2) Found counterexample ab for current model $\mathcal{R}_{\mathcal{T}_1}$ (cf. Figure B.5(a)).
3) $\mathcal{T}_2$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row a to U.
4) $\mathcal{T}_3$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row ba to U.
5) Found counterexample ba for current model $\mathcal{R}_{\mathcal{T}_4}$ (cf. Figure B.5(b)).
6) $\mathcal{T}_5$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row bab to U.
7) Table $\mathcal{T}_6$ is RFSA-closed and RFSA-consistent, and the final model $\mathcal{R}_{\mathcal{T}_6}$ is calculated (cf. Figure B.5(c)).

Table B.5: An example of an NL* run where an intermediate hypothesis is not an RFSA

B.4 An example for non-termination of a row-based algorithm

In this section, we present an example demonstrating that the RFSA algorithm would not terminate if we added counterexamples and their prefixes to the set of rows U instead of adding counterexamples and their suffixes to the set of columns V. The problem we face with this version of the RFSA learning algorithm is that there are (rare) cases where a word $u \in U$ does not lead to the state $row(u)$. Therefore, the algorithm detects u as a counterexample, though it is already contained and classified in the table. The RFSA-closed and RFSA-consistent table $\mathcal{T}_3$, for example, classifies word a as positive, but the NFA of $\mathcal{T}_3$ does not accept word a, resulting in a non-terminating cycle within the inference algorithm.



(a) Minimal DFA for the regular language to infer

(b) NFA $\mathcal{R}_{\mathcal{T}_3}$

Figure B.6: DFA, NFA: An example for non-termination of algorithm from Section 4.5

	$\mathcal{T}_0$	ε
$*$	ε	$+$
$*$	b	$+$
$*$	a	$+$

$\Longrightarrow^{1.}_{ce}$

	$\mathcal{T}_1$	ε
$*$	ε	$+$
$*$	a	$+$
$*$	aa	$-$
$*$	b	$+$
$*$	ab	$-$
$*$	aab	$-$
$*$	aaa	$-$

$\Longrightarrow^{2.}_{ncs}$

	$\mathcal{T}_2$	ε	b
	ε	$+$	$+$
$*$	a	$+$	$-$
$*$	aa	$-$	$-$
	b	$+$	$+$
$*$	ab	$-$	$+$
$*$	aab	$-$	$-$
$*$	aaa	$-$	$-$

$\Longrightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	b
	ε	$+$	$+$
$*$	a	$+$	$-$
$*$	aa	$-$	$-$
$*$	ab	$-$	$+$
	b	$+$	$+$
$*$	aab	$-$	$-$
$*$	aaa	$-$	$-$
	abb	$+$	$+$
$*$	aba	$-$	$-$

1) Found counterexample aa for the current model $\mathcal{R}_{\mathcal{T}_0}$ based on $\mathcal{T}_0$.
2) $\mathcal{T}_1$ is not RFSA-consistent (cf. Definition 4.5.2). Trying to obtain weak RFSA-consistency by adding suffix b to V.
3) $\mathcal{T}_2$ is not RFSA-closed. Trying to obtain RFSA-closedness by adding row ab to U.
4) Trying to add counterexample $a \in L(\mathcal{A}) \setminus L(\mathcal{R}_{\mathcal{T}_3})$ fails, because a is already present in $\mathcal{T}_3$ but not accepted by hypothesis $\mathcal{R}_{\mathcal{T}_3}$ (cf. Figures B.6(a) and B.6(b)).

Table B.6: Non-termination problem with the algorithm that adds counterexamples to U

B.5 An example for a non-increasing number of states

Let us consider an example where the number of states does not increase after inserting a counterexample.

In contrast to Angluin's learning algorithm L*, in NL* it might be the case that adding a new counterexample to the table does not introduce a new state. As shown in Theorem 4.3.3, the termination of the algorithm can still be assured though it is substantially more involved than in the case of L*.

Let $\Sigma = \{a, b, c\}$. We want to infer the regular language of the minimal DFA from Figure B.7. The corresponding canonical RFSA $\mathcal{R}_{\mathcal{T}_8}$ and an intermediate hypothesis $\mathcal{R}_{\mathcal{T}_6}$ are depicted in Figure B.8.

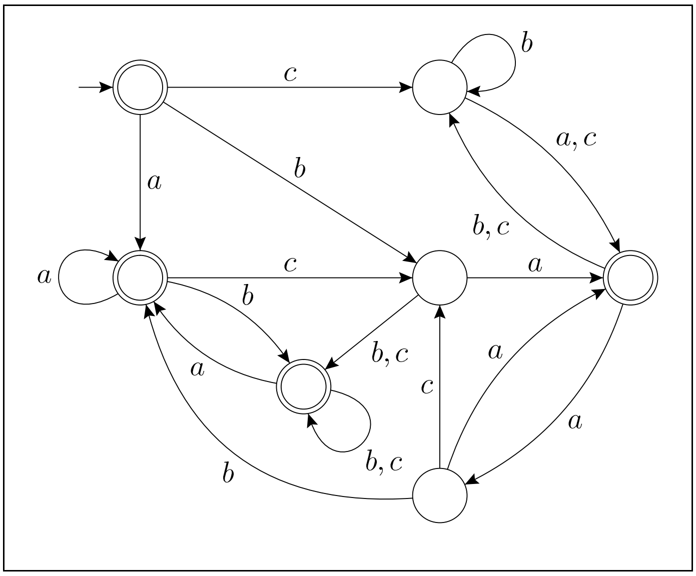
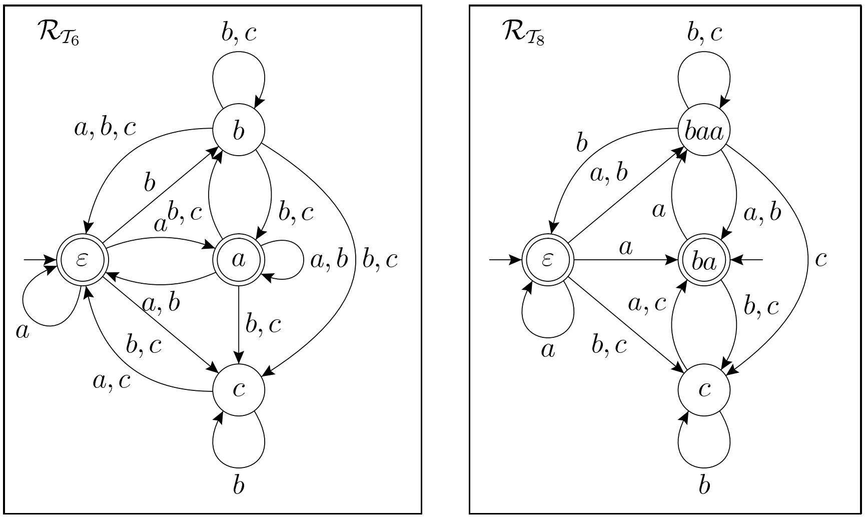

b
c
a, c
b
a
b, c
a
c
a
b
b, c
a
a
c
a
b, c
b

Figure B.7: The minimal DFA corresponding to RFSA $\mathcal{R}_{\mathcal{T}_8}$ from Figure B.8


$\mathcal{R}_{\mathcal{T}_6}$
b, c
a, b, c
b
b
a
b, c
b, c
ε
a
a, b
b, c
a, b
a
b, c
b, c
a, c
c
b
$\mathcal{R}_{\mathcal{T}_8}$
b, c
b
baa
a, b
a
a, b
ε
a
ba
c
a, c
b, c
a
b, c
c
b

(a) RFSA $\mathcal{R}_{\mathcal{T}_6}$ derived from table $\mathcal{T}_6$ (b) RFSA $\mathcal{R}_{\mathcal{T}_8}$ derived from table $\mathcal{T}_8$

Figure B.8: An example for a non-increasing number of states

	$\mathcal{T}_0$	ε
*	ε	+
*	b	−
*	c	−
*	a	+

$\Rightarrow^{1.}_{ncl}$

	$\mathcal{T}_1$	ε
*	ε	+
*	b	−
*	c	−
*	a	+
*	bb	+
*	bc	+
*	ba	+

$\Rightarrow^{2.}_{ncs}$

	$\mathcal{T}_2$	ε	b
*	ε	+	−
*	b	−	+
*	c	−	−
	a	+	+
	bb	+	+
	bc	+	+
*	ba	+	−

$\Rightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	b
*	ε	+	−
*	b	−	+
*	c	−	−
	a	+	+
	bb	+	+
	bc	+	+
*	ba	+	−
*	cb	−	−
*	cc	+	−
*	ca	+	−

$\Rightarrow^{4.}_{ncs}$

	$\mathcal{T}_4$	ε	b	c
*	ε	+	−	−
*	b	−	+	+
*	c	−	−	+
*	a	+	+	−
	bb	+	+	+
	bc	+	+	+
*	ba	+	−	−
*	cb	−	−	+
*	cc	+	−	−
*	ca	+	−	−

$\Rightarrow^{5.}_{ncl}$

	$\mathcal{T}_5$	ε	b	c
*	ε	+	−	−
*	b	−	+	+
*	c	−	−	+
*	a	+	+	−
	bb	+	+	+
	bc	+	+	+
*	ba	+	−	−
*	cb	−	−	+
*	cc	+	−	−
*	ca	+	−	−
	ab	+	+	+
*	ac	−	+	+
*	aa	+	+	−

$\Rightarrow^{6.}_{ce}$

	$\mathcal{T}_6$	ε	b	c	baa	aa	a
*	ε	+	−	−	−	+	+
*	b	−	+	+	+	−	+
*	c	−	−	+	−	−	+
*	a	+	+	−	+	+	+
	bb	+	+	+	+	+	+
	bc	+	+	+	+	+	+
*	ba	+	−	−	−	+	−
*	cb	−	−	+	−	−	+
*	cc	+	−	−	−	+	−
*	ca	+	−	−	−	+	−
	ab	+	+	+	+	+	+
*	ac	−	+	+	+	−	+
*	aa	+	+	−	+	+	+

$\Rightarrow^{7.}_{ncl}$

	$\mathcal{T}_7$	ε	b	c	baa	aa	a
*	ε	+	−	−	−	+	+
	b	−	+	+	+	−	+
*	c	−	−	+	−	−	+
	a	+	+	−	+	+	+
*	ba	+	−	−	−	+	−
	bb	+	+	+	+	+	+
	bc	+	+	+	+	+	+
*	cb	−	−	+	−	−	+
*	cc	+	−	−	−	+	−
*	ca	+	−	−	−	+	−
	ab	+	+	+	+	+	+
	ac	−	+	+	+	−	+
	aa	+	+	−	+	+	+
*	bab	−	−	+	−	−	+
*	bac	−	−	+	−	−	+
*	baa	−	+	−	+	−	+

$\Rightarrow^{8.}_{ncl}$

	$\mathcal{T}_8$	ε	b	c	baa	aa	a
*	ε	+	−	−	−	+	+
	b	−	+	+	+	−	+
*	c	−	−	+	−	−	+
	a	+	+	−	+	+	+
*	ba	+	−	−	−	+	−
*	baa	−	+	−	+	−	+
	bb	+	+	+	+	+	+
	bc	+	+	+	+	+	+
*	cb	−	−	+	−	−	+
*	cc	+	−	−	−	+	−
*	ca	+	−	−	−	+	−
	ab	+	+	+	+	+	+
	ac	−	+	+	+	−	+
	aa	+	+	−	+	+	+
*	bab	−	−	+	−	−	+
*	bac	−	−	+	−	−	+
	$baab$	+	+	−	+	+	+
	$baac$	−	+	+	+	−	+
*	$baaa$	+	−	−	−	+	−

1) $\mathcal{T}_0$ is not RFSA-closed. Hence, we try to make it RFSA-closed by moving row b to the upper table.

2) $\mathcal{T}_1$ violates the RFSA-consistency property, as $b \sqsubseteq \varepsilon$ but $bb \not\sqsubseteq b$. We try to obtain RFSA-consistency by adding suffix b to V.

3) After resolving the inconsistency the table is not RFSA-closed. Thus, we add row c to U.

4) Table $\mathcal{T}_3$ is inconsistent again ($c \sqsubseteq \varepsilon$ but $cc \not\sqsubseteq c$) and we insert suffix c to V.

5) An RFSA-closedness violation forces us to move row a to U.

6) As table $\mathcal{T}_5$ is RFSA-closed and RFSA-consistent, a hypothesis can be derived (cf. Figure B.8(a)) and a counterexample is found. The word baa is accepted by our hypothesis but not contained in the language to learn. Hence, we add $\mathit{suff}(baa)$ to V.

7)−9) Resolving two more RFSA-closedness violations by moving rows ba and baa to the upper table, table $\mathcal{T}_8$ is RFSA-closed and RFSA-consistent, again. The final model $\mathcal{R}_{\mathcal{T}_8}$ is calculated and depicted in Figure B.8(b).

Table B.7: An example of an NL* run where the number of states does not increase

B.6 An example for a decreasing number of states

In the previous chapter we already saw an example showing that hypotheses do not necessarily have to increase. In this section, we furthermore provide an example where the number of states from one hypothesis to the next hypothesis even decreases. Let us consider the regular language L recognized by the minimal DFA from Figure B.9. While learning L, the intermediate NFA $\mathcal{R}_{\mathcal{T}_7}$ is derived from table $\mathcal{T}_7$ (cf. Table B.8). It contains six states whereas the next hypothesis (derived from table $\mathcal{T}_{10}$) has only five states. The reason for this is that—due to an RFSA-closedness violation—from table $\mathcal{T}_7$ to $\mathcal{T}_{10}$ $row(aab)$ is added to the upper table which helps to compose the previous prime states $row(a)$ and $row(b)$.

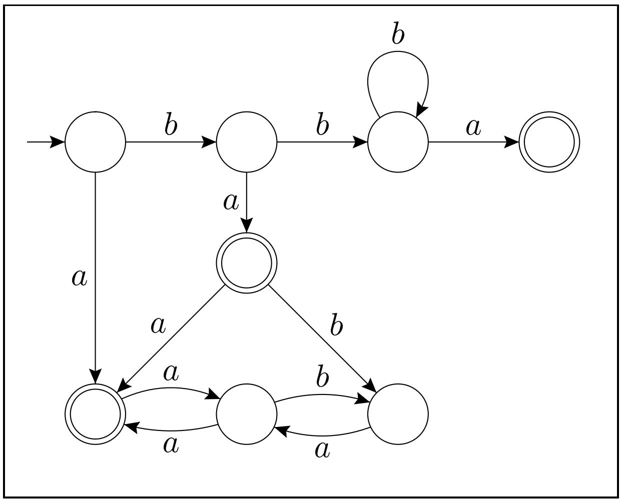
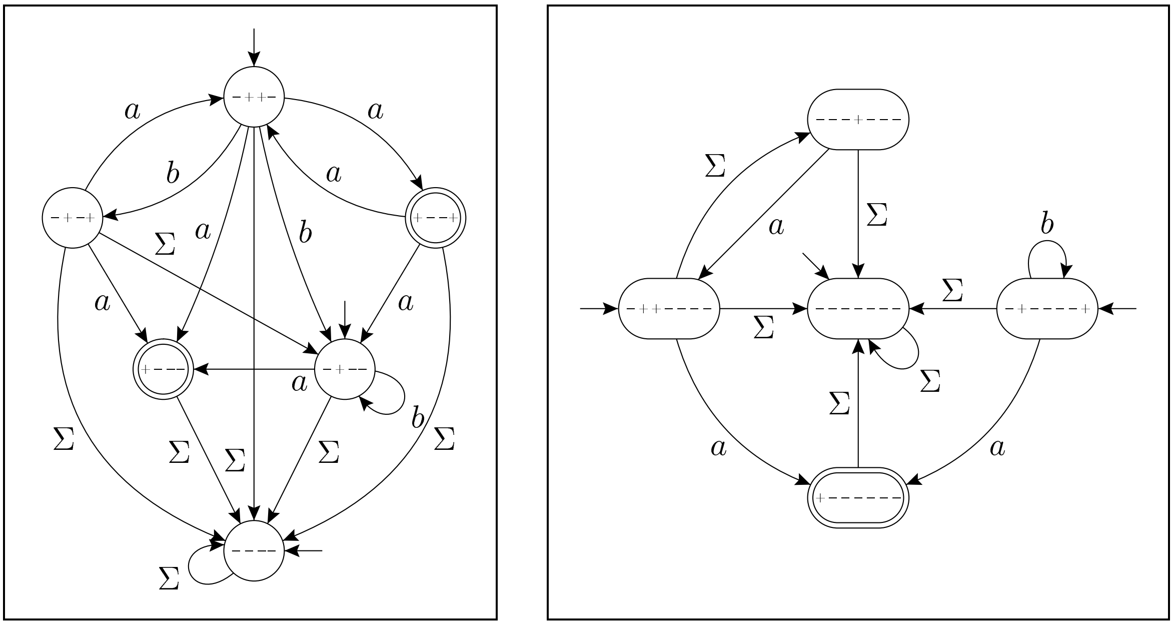


Figure B.9: Minimal DFA for the regular language to learn



(a) Automaton $\mathcal{R}_{\mathcal{T}_7}$ for table $\mathcal{T}_7$

(b) Automaton $\mathcal{R}_{\mathcal{T}_{10}}$ for table $\mathcal{T}_{10}$

Figure B.10: Two successive hypotheses with decreasing state number

	$\mathcal{T}_0$	ε
*	ε	−
*	b	−
*	a	+

$\Longrightarrow^{1.}_{ncl}$

	$\mathcal{T}_1$	ε
*	ε	−
*	a	+
*	b	−
*	ab	−
*	aa	−

$\Longrightarrow^{2.}_{ncs}$

	$\mathcal{T}_2$	ε	a
*	ε	−	+
*	a	+	−
*	b	−	+
*	ab	−	−
*	aa	−	+

$\Longrightarrow^{3.}_{ncl}$

	$\mathcal{T}_3$	ε	a
*	ε	−	+
*	a	+	−
*	ab	−	−
*	b	−	+
*	aa	−	+
*	abb	−	−
*	aba	−	−

$\Longrightarrow^{4.}_{ce}$

	$\mathcal{T}_4$	ε	a	baa	aa
*	ε	−	+	+	−
*	a	+	−	−	+
*	ab	−	−	−	−
*	b	−	+	−	+
*	aa	−	+	+	−
*	abb	−	−	−	−
*	aba	−	−	−	−

$\Longrightarrow^{5.}_{ncl}$

	$\mathcal{T}_5$	ε	a	baa	aa
*	ε	−	+	+	−
*	b	−	+	−	+
*	a	+	−	−	+
*	ab	−	−	−	−
*	aa	−	+	+	−
*	abb	−	−	−	−
*	aba	−	−	−	−
*	bb	−	+	−	−
*	ba	+	+	+	−

$\Longrightarrow^{6.}_{ncl}$

	$\mathcal{T}_6$	ε	a	baa	aa
*	ε	−	+	+	−
*	b	−	+	−	+
*	a	+	−	−	+
*	ab	−	−	−	−
*	bb	−	+	−	−
*	aa	−	+	+	−
*	abb	−	−	−	−
*	aba	−	−	−	−
	ba	+	+	+	−
*	bbb	−	+	−	−
*	bba	+	−	−	−

$\Longrightarrow^{7.}_{ncl}$

	$\mathcal{T}_7$	ε	a	baa	aa
*	ε	−	+	+	−
*	b	−	+	−	+
*	a	+	−	−	+
*	ab	−	−	−	−
*	bb	−	+	−	−
*	bba	+	−	−	−
*	aa	−	+	+	−
*	abb	−	−	−	−
*	aba	−	−	−	−
	ba	+	+	+	−
*	bbb	−	+	−	−
*	$bbab$	−	−	−	−
*	$bbaa$	−	−	−	−

$\Longrightarrow^{8.}_{ce}$

	$\mathcal{T}_8$	ε	a	baa	aa	$baba$	aba	ba
	ε	−	+	+	−	−	−	+
*	b	−	+	−	+	−	−	+
*	a	+	−	−	+	−	−	−
*	ab	−	−	−	−	−	−	−
*	bb	−	+	−	−	−	−	+
*	bba	+	−	−	−	−	−	−
*	aa	−	+	+	−	−	−	−
*	abb	−	−	−	−	−	−	−
*	aba	−	−	−	−	−	−	−
	ba	+	+	+	−	−	−	−
*	bbb	−	+	−	−	−	−	+
*	$bbab$	−	−	−	−	−	−	−
*	$bbaa$	−	−	−	−	−	−	−

$\Longrightarrow^{9.}_{ncl}$

	$\mathcal{T}_9$	ε	a	baa	aa	$baba$	aba	ba
	ε	−	+	+	−	−	−	+
	b	−	+	−	+	−	−	+
	a	+	−	−	+	−	−	−
*	ab	−	−	−	−	−	−	−
*	aa	−	+	+	−	−	−	−
*	bb	−	+	−	−	−	−	+
*	bba	+	−	−	−	−	−	−
*	abb	−	−	−	−	−	−	−
*	aba	−	−	−	−	−	−	−
	ba	+	+	+	−	−	−	−
*	bbb	−	+	−	−	−	−	+
*	$bbab$	−	−	−	−	−	−	−
*	$bbaa$	−	−	−	−	−	−	−
*	aab	−	−	−	+	−	−	−
	aaa	+	−	−	+	−	−	−

$\Longrightarrow^{10.}_{ncl}$

	$\mathcal{T}_{10}$	ε	a	baa	aa	$baba$	aba	ba
	ε	−	+	+	−	−	−	+
	b	−	+	−	+	−	−	+
	a	+	−	−	+	−	−	−
*	ab	−	−	−	−	−	−	−
*	aa	−	+	+	−	−	−	−
*	bb	−	+	−	−	−	−	+
*	bba	+	−	−	−	−	−	−
*	aab	−	−	−	+	−	−	−
*	abb	−	−	−	−	−	−	−
*	aba	−	−	−	−	−	−	−
	ba	+	+	+	−	−	−	−
*	bbb	−	+	−	−	−	−	+
*	$bbab$	−	−	−	−	−	−	−
*	$bbaa$	−	−	−	−	−	−	−
	aaa	+	−	−	+	−	−	−
*	$aabb$	−	−	−	−	−	−	−
*	$aaba$	−	+	+	−	−	−	−

1) $\mathcal{T}_0$ is not RFSA-closed. Hence, we try to make it RFSA-closed by moving row a to the upper table.

2) $\mathcal{T}_1$ violates the RFSA-consistency property, as $\varepsilon \sqsubseteq a$ but $a \not\sqsubseteq aa$. We try to obtain RFSA-consistency by adding suffix a to V.

3) After resolving the inconsistency the table is not RFSA-closed. Thus, we add row ab to U.

4) The table is RFSA-closed and RFSA-consistent but a counterexample is found. The word baa is not accepted by our hypothesis but contained in the language to learn. Hence, we add $\mathit{suff}(baa)$ to V.

5) – 7) RFSA-closedness violations force us to move rows b, bb, and bba to U.

8) Again the table ($\mathcal{T}_7$) is RFSA-closed and RFSA-consistent yielding the intermediate hypothesis $\mathcal{R}_{\mathcal{T}_7}$ for which a counterexample can be derived. The word $baba$ is accepted by our hypothesis (cf. Figure B.10(a)) but not contained in the language to learn. Hence, we add $\mathit{suff}(baba)$ to V.

9) – 11) Two more RFSA-closedness violations force us to move rows aa and aab to U yielding an RFSA-closed and RFSA-consistent table $\mathcal{T}_{10}$. The final model $\mathcal{R}_{\mathcal{T}_{10}}$ is calculated and depicted in Figure B.10(b).

Table B.8: An example of an NL* run where the number of states even decreases

B.7 An example where NL^* needs more equivalence queries than states in minimal DFA

A very interesting case is presented in this section where NL* learns a regular language for which it needs more equivalence queries than the corresponding minimal DFA has got states. Note that this can never happen in the L* algorithm, as it is guaranteed that L* needs at most n equivalence queries. In the example the minimal DFA (cf. Figure B.11) contains 5 states and NL* needs in total 6 equivalence queries.

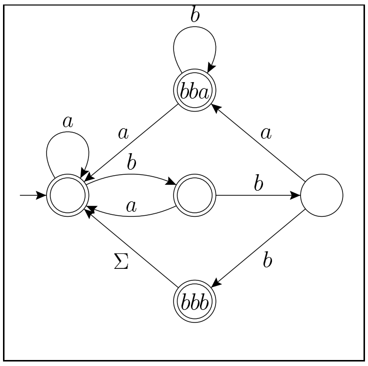
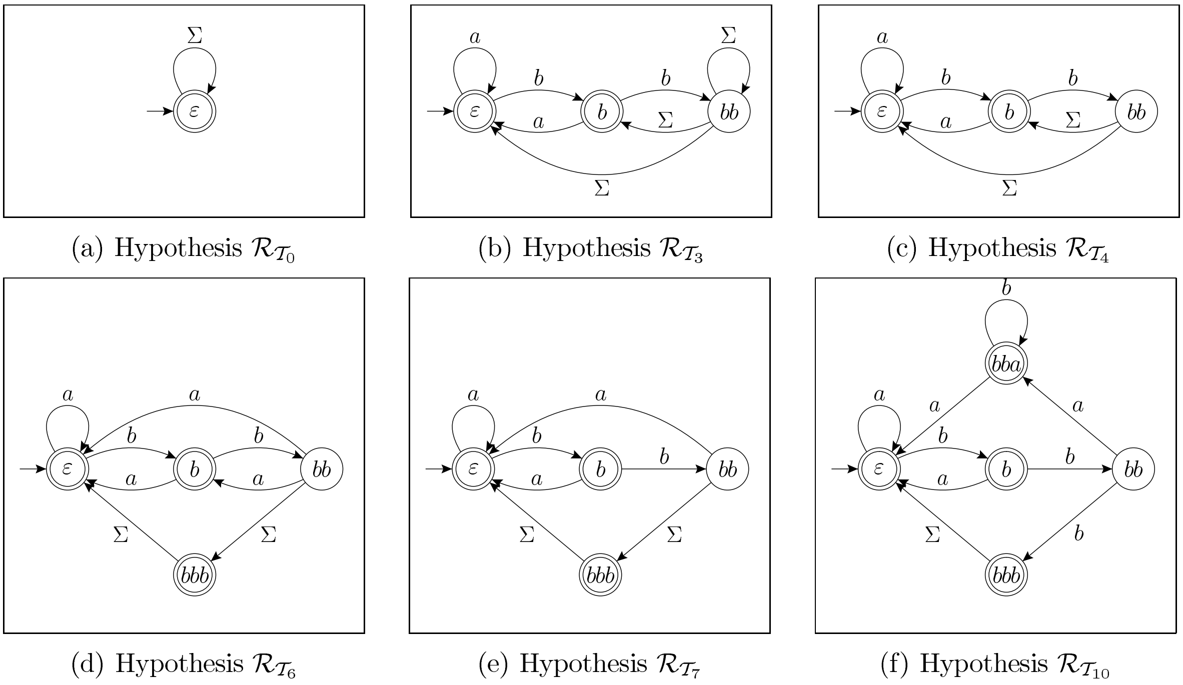


Figure B.11: Minimal DFA recognizing the language to infer



(a) Hypothesis $\mathcal{R}_{\mathcal{T}_0}$

(b) Hypothesis $\mathcal{R}_{\mathcal{T}_3}$

(c) Hypothesis $\mathcal{R}_{\mathcal{T}_4}$

(d) Hypothesis $\mathcal{R}_{\mathcal{T}_6}$

(e) Hypothesis $\mathcal{R}_{\mathcal{T}_7}$

(f) Hypothesis $\mathcal{R}_{\mathcal{T}_{10}}$

Figure B.12: All hypotheses for inferring the canonical RFSA for minimal DFA from Figure B.11

	$\mathcal{T}_0$	ε
$*$	ε	$+$
$*$	b	$+$
$*$	a	$+$

$\Longrightarrow$ 1. *ce*

	$\mathcal{T}_1$	ε	bb	b
$*$	ε	$+$	$-$	$+$
$*$	b	$+$	$+$	$-$
$*$	a	$+$	$-$	$+$

$\Longrightarrow$ 2. *ncl*

	$\mathcal{T}_2$	ε	bb	b
$*$	ε	$+$	$-$	$+$
$*$	b	$+$	$+$	$-$
$*$	a	$+$	$-$	$+$
$*$	bb	$-$	$+$	$+$
$*$	ba	$+$	$-$	$+$

$\Longrightarrow$ 3. *ncl*

	$\mathcal{T}_3$	ε	bb	b
$*$	ε	$+$	$-$	$+$
$*$	b	$+$	$+$	$-$
$*$	bb	$-$	$+$	$+$
$*$	a	$+$	$-$	$+$
$*$	ba	$+$	$-$	$+$
	bbb	$+$	$+$	$+$
	bba	$+$	$+$	$+$

$\Longrightarrow$ 4. *ce*

	$\mathcal{T}_4$	ε	bb	b	$bbaabb$	$baabb$	$aabb$	abb
$*$	ε	$+$	$-$	$+$	$-$	$-$	$-$	$-$
$*$	b	$+$	$+$	$-$	$-$	$-$	$-$	$-$
$*$	bb	$-$	$+$	$+$	$-$	$-$	$-$	$+$
$*$	a	$+$	$-$	$+$	$-$	$-$	$-$	$-$
$*$	ba	$+$	$-$	$+$	$-$	$-$	$-$	$-$
	bbb	$+$	$+$	$+$	$-$	$-$	$-$	$-$
	bba	$+$	$+$	$+$	$-$	$-$	$-$	$-$

$\Longrightarrow$ 5. *ce*

	$\mathcal{T}_5$	ε	bb	b	$bbaabb$	$baabb$	$aabb$	abb	$bbbbbb$	$bbbbb$	$bbbb$	bbb
$*$	ε	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	b	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$+$
$*$	bb	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$+$
$*$	a	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	ba	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	bbb	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$
	bba	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$+$

$\Longrightarrow$ 6. *ncl*

	$\mathcal{T}_6$	ε	bb	b	$bbaabb$	$baabb$	$aabb$	abb	$bbbbbb$	$bbbbb$	$bbbb$	bbb
$*$	ε	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	b	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$+$
$*$	bb	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$+$
$*$	bbb	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$
$*$	a	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	ba	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
	bba	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$+$
$*$	$bbbb$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	$bbba$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$

$\Longrightarrow$ 7. *ce*

	$\mathcal{T}_7$	ε	bb	b	$bbaabb$	$baabb$	$aabb$	abb	$bbbbbb$	$bbbbb$	$bbbb$	bbb	$bbababb$	$bababb$	$ababb$	$babb$
$*$	ε	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
$*$	b	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$+$	$-$	$-$	$-$	$+$
$*$	bb	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$+$	$-$	$-$	$-$	$-$
$*$	bbb	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$
$*$	a	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
$*$	ba	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
	bba	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
$*$	$bbbb$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$
$*$	$bbba$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$

$\Longrightarrow$ 8. *ce*

	$\mathcal{T}_8$	ε	bb	b	$bbaabb$	$baabb$	$aabb$	abb	$bbbbbb$	$bbbbb$	$bbbb$	bbb	$bbababb$	$bababb$	$ababb$	$babb$	$bbabbabb$	$babbabb$	$abbabb$	$bbabb$
$*$	ε	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	b	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$+$	$-$	$+$	$-$
$*$	bb	$-$	$+$	$+$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$-$	$-$
$*$	bbb	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$
$*$	a	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	ba	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	bba	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$
$*$	$bbbb$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$
$*$	$bbba$	$+$	$-$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$+$

$\Longrightarrow$ 9. *ncl*

	$\mathcal{T}_9$	ε	*bb*	*b*	*bbaabb*	*baabb*	*aabb*	*abb*	*bbbbbb*	*bbbbb*	*bbbb*	*bbb*	*bbababb*	*bababb*	*ababb*	*babb*	*bbabbabb*	*babbabb*	*abbabb*	*bbabb*
$*$	ε	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+
$*$	*b*	+	+	−	−	−	−	−	+	−	+	+	−	−	−	+	+	−	+	−
$*$	*bb*	−	+	+	−	−	−	+	+	+	−	+	−	−	−	−	+	+	−	−
$*$	*bbb*	+	+	+	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*bba*	+	+	+	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	−
$*$	*a*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+
$*$	*ba*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+
$*$	*bbbb*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+
$*$	*bbba*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+
$*$	*bbab*	+	+	+	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	−
$*$	*bbaa*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+

$\Longrightarrow$ 10. $ncs1$

	$\mathcal{T}_{10}$	ε	*bb*	*b*	*bbaabb*	*baabb*	*aabb*	*abb*	*bbbbbb*	*bbbbb*	*bbbb*	*bbb*	*bbababb*	*bababb*	*ababb*	*babb*	*bbabbabb*	*babbabb*	*abbabb*	*bbabb*	*bbbabb*
$*$	ε	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*b*	+	+	−	−	−	−	−	+	−	+	+	−	−	−	+	+	−	+	−	−
$*$	*bb*	−	+	+	−	−	−	+	+	+	−	+	−	−	−	−	+	+	−	−	−
$*$	*bbb*	+	+	+	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−	+
$*$	*bba*	+	+	+	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	−	−
$*$	*a*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*ba*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*bbbb*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*bbba*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−
$*$	*bbab*	+	+	+	−	−	−	−	+	+	+	+	−	−	−	−	+	+	+	−	−
$*$	*bbaa*	+	−	+	−	−	−	−	−	+	+	+	−	−	−	−	−	+	+	+	−

1) Found counterexample *bb* for current model $\mathcal{R}_{\mathcal{T}_0}$ (cf. Figure B.12(a)).
2) $\mathcal{T}_1$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row *b* to U.
3) $\mathcal{T}_2$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row *bb* to U.
4) Found counterexample *bbaabb* for current model $\mathcal{R}_{\mathcal{T}_3}$ (cf. Figure B.12(b)).
5) Found counterexample *bbbbbb* for current model $\mathcal{R}_{\mathcal{T}_4}$ (cf. Figure B.12(c)).
6) $\mathcal{T}_5$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row *bbb* to U.
7) Found counterexample *bbababb* for current model $\mathcal{R}_{\mathcal{T}_6}$ (cf. Figure B.12(d)).
8) Found counterexample *bbabbabb* for current model $\mathcal{R}_{\mathcal{T}_7}$ (cf. Figure B.12(e)).
9) $\mathcal{T}_8$ is not RFSA-closed. Trying to obtain RFSA-closedness by moving row *bba* to U.
10) $\mathcal{T}_9$ violates the RFSA-consistency property. We try to obtain RFSA-consistency by adding suffix *bbbabb* to V.
11) Table $\mathcal{T}_{10}$ is RFSA-closed and RFSA-consistent. The final model $\mathcal{R}_{\mathcal{T}_{10}}$ is calculated and depicted in Figure B.12(f)).

Table B.9: An example where NL* needs more equivalence queries than the corresponding minimal DFA contains states

C An Algorithm for Solving the PDL Membership Problem

Local formula check:

```
V = {0, .. , n-1}

boolean[ ] Sat(LocalFormula f) {
    boolean[ ] sat = new boolean[n];
    switch(f) {
    case Not(f1):
        boolean[ ] sat1 = Sat(f1);
        for (int i = 0; i < n; i++)
            sat[i] = !sat1[i];
        break;
    case Or(f1, f2):
        boolean[ ] sat1 = Sat(f1);
        boolean[ ] sat2 = Sat(f2);
        for (int i = 0; i < n; i++)
            sat[i] = sat1[i] || sat2[i];
        break;
    case Event(..):
        for (int i = 0; i < n; i++)
            sat[i] = (V[i].event.equals(f));
        break;
    case <p1> f2:
        boolean[ ][ ] trans1 = Trans(p1);
        boolean[ ] sat2 = Sat(f2);
        for (int i = 0; i < n; i++) {
            sat[i] = false;
            for (int j = 0; j < n; j++)
                if(trans[i][j])
                sat[i] = sat2[j];
        }
        break;
    case <p1>$^{-1}$ f2:
        boolean[ ][ ] trans1 = TransBack(p1);
        boolean[ ] sat2 = Sat(f2);
        for (int i = 0; i < n; i++) {
            sat[i] = false;
            for (int j = 0; j < n; j++)
                if(trans[i][j])
                sat[i] = sat2[j];
        }
        break;
    }
}
```

Table C.1: Algorithm for checking local formulas

Forward path expression check:

```
boolean[ ][ ] Trans(PathFormula p) {
    boolean[ ][ ] trans = new boolean[n][n];
    switch(p) {
    case (p1; p2):
        boolean[ ][ ] trans1 = Trans(p1);
        boolean[ ][ ] trans2 = Trans(p2);
        for (int i = 0; i < n; i++)
            for (int k = 0; k < n; k++) {
                trans[i][k] = false;
                for (int j = 0; j < n; j++)
                    if(trans1[i][j] && trans1[j][k])
                        trans[i][k] = true;
            }
        break;
    case p1 + p2:
        boolean[ ][ ] trans1 = Trans(p1);
        boolean[ ][ ] trans2 = Trans(p2);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                trans[i][j] = trans1[i][j] || trans2[i][j];
        break;
    case p1*:
        boolean[ ][ ] trans1 = Trans(p1);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                star[i][j] = (i==j);
        while (true) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    if (trans1[i][j])
                        for (int k = 0; k < n; k++)
                            if (!trans[i][k] && trans1[j][k]) {
                                trans[i][k] = true;
                                continue;
                            }
            break;
        }
        break;
    }
}
```

Table C.2: Algorithm for checking forward path expressions

Backward path expression check:

```
boolean[ ][ ] TransBack(PathFormula p) {
    boolean[ ][ ] transBack = new boolean[n][n];
    switch(p) {
    case (p1; p2):
        boolean[ ][ ] transBack1 = TransBack(p1);
        boolean[ ][ ] transBack2 = TransBack(p2);
        for (int i = 0; i < n; i++)
            for (int k = 0; k < n; k++) {
                transBack[i][k] = false;
                for (int j = 0; j < n; j++)
                    if(transBack1[i][j] && transBack1[j][k])
                        transBack[i][k] = true;
            }
        break;
    case p1 + p2:
        boolean[ ][ ] transBack1 = TransBack(p1);
        boolean[ ][ ] transBack2 = TransBack(p2);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                transBack[i][j] = transBack1[i][j] || transBack2[i][j];
        break;
    case p1*:
        boolean[ ][ ] transBack1 = TransBack(p1);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                star[i][j] = (i==j);
        while (true) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    if (transBack1[i][j])
                        for (int k = 0; k < n; k++)
                            if (!transBack[i][k] && transBack1[j][k]) {
                                transBack[i][k] = true;
                                continue;
                            }
            break;
        }
        break;
    }
}
```

Table C.3: Algorithm for checking backward path expressions

Bibliography

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Realizability and Verification of MSC Graphs. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), Crete, Greece*, volume 2076 of *Lecture Notes in Computer Science*. Springer, 2001.

[AEY03] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[AEY05] R. Alur, K. Etessami, and M. Yannakakis. Realizability and Verification of MSC Graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[AHP96] R. Alur, G. J. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[AJ02] S. W. Ambler and R. Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, 2002.

[AJNS04] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A Survey of Regular Model Checking. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004), London, UK*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[AMKN05] B. Adsul, M. Mukund, K. Narayan Kumar, and V. Narayanan. Causal Closure for MSC Languages. In R. Ramanujam and S. Sen, editors, *Proceedings of the 25st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2005), Hyderabad, India*, volume 3821 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2005.

[Amn03] T. Amnell. *Code Synthesis for Timed Automata*. PhD thesis, Uppsala University, Sweden, 2003.

[Ang87a] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Ang87b] D. Angluin. Queries and Concept Learning. *Machine Learning*, 2(4):319–342, 1987.

[Ara98] J. Araújo. Formalizing Sequence Diagrams. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, volume 33(10) of *ACM SIGPLAN Notices*. ACM, 1998.

[Ard60] D. N. Arden. Delayed-Logic and Finite-State Machines. In *Theory of Computing Machine Design*, pages 1–35. University of Michigan Press, 1960.

[Ard61] D. N. Arden. Delayed-Logic and Finite-State Machines. In *Proceedings of the Second Annual Symposium and Papers from the First Annual Symposium on Switching Circuit Theory and Logical Design (FOCS 1961), Detroit, Michigan, USA*, pages 133–151. American Institute of Electrical Engineers, 1961.

[AT&T] AT&T. Grappa - A `Java` Graph Package. Library available at: `http://www.research.att.com/~john/Grappa/`.

[BAL97] H. Ben-Abdallah and S. Leue. Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts. In E. Brinksma, editor, *Proceedings of the 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1997), Enschede, The Netherlands*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 1997.

[BAL98] H. Ben-Abdallah and S. Leue. MESA: Support for Scenario-Based Design of Concurrent Systems. In B. Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1998), Lisbon, Portugal*, volume 1384 of *Lecture Notes in Computer Science*, pages 118–135. Springer, 1998.

[BBH06] D. Babic, J. D. Bingham, and A. J. Hu. B-Cubing: New Possibilities for Efficient SAT-Solving. *IEEE Transactions on Computers*, 55(11):1315–1324, 2006.

[BCG$^+$99] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. L. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of Software Programs for Embedded Control Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, 1999.

[BF72] A. W. Biermann and J. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Software Engineering*, 21(6):592–597, 1972.

[BFS04] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for Test Case Generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer, 2004.

[BGJ$^+$05] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), Edinburgh, UK*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

[BHKL08] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. Research Report LSV-08-28, Laboratoire Spécification et Vérification, ENS Cachan, France, October 2008. 30 pages.

[BHKL09] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In C. Boutilier, editor, *Proceedings of the 21th International Joint Conference on Artificial Intelligence (IJCAI 2009), Pasadena, California, USA*, pages 1004–1009. AAAI Press, 2009.

[BJK+05] M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000), Chicago, Illinois, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[BK08] C. Baier and J. P. Katoen. *Principles of Model Checking.* The MIT Press, May 2008.

[BKKL] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering.* To appear.

[BKKL06] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. Replaying Play In and Play Out: Synthesis of Design Models from Scenarios by Learning. Research Report AIB-2006-12, RWTH Aachen University, Germany, 2006. 28 pages.

[BKKL07] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. Replaying Play In and Play Out: Synthesis of Design Models from Scenarios by Learning. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), Braga, Portugal*, volume 4424 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2007.

[BKKL08a] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. *Smyle*: A Tool for Synthesizing Distributed Models from Scenarios by Learning. In F. van Breugel and M. Chechik, editors, *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008), Toronto, Canada*, volume 5201 of *Lecture Notes in Computer Science*, pages 162–166. Springer, 2008.

[BKKL08b] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. SMA—The Smyle Modeling Approach. Technical Report TUM-I0820, TU München, Germany, 2008. 26 pages.

[BKKL09] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. SMA—The Smyle Modeling Approach. In Z. Huzar and B. Meyer, editors, *Proceedings of the 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2008), Brno, Czech Republic*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[BKKL10] B. Bollig, J. P. Katoen, C. Kern, and M. Leucker. SMA—The Smyle Modeling Approach. *Computing and Informatics*, (1), 2010. To appear.

[BKM07] B. Bollig, D. Kuske, and I. Meinecke. Propositional Dynamic Logic for Message-Passing Systems. In V. Arvind and S. Prasad, editors, *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007), New Delhi, India*, volume 4855 of *Lecture Notes in Computer Science*, pages 303–315. Springer, 2007.

[BKSS06] B. Bollig, C. Kern, M. Schlütter, and V. Stolz. MSCan: A Tool for Analyzing MSC Specifications. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006), Vienna, Austria*, volume 3920 of *Lecture Notes in Computer Science*, pages 455–458. Springer, 2006.

[BL05] B. Bollig and M. Leucker. A Hierarchy of Implementable MSC Languages. In F. Wang, editor, *Proceedings of the 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005), Taipei, Taiwan*, volume 3731 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2005.

[Blu09] Specification of the Bluetooth System (version 3.0), April 2009. `http://bluetooth.com/Bluetooth/Technology/Building/Specifications/`.

[BM03] N. Baudru and R. Morin. Safe Implementability of Regular Message Sequence Chart Specifications. In W. Dosch and R. Y. Lee, editors, *Proceedings of the ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2003), Lübeck, Germany*, volume 2380 of *Lecture Notes in Computer Science*, pages 210–217. Springer, 2003.

[BM07] N. Baudru and R. Morin. Synthesis of Safe Message-Passing Systems. In V. Arvind and S. Prasad, editors, *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007), New Delhi, India*, volume 1664 of *Lecture Notes in Computer Science*, pages 277–289. Springer, 2007.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[BSL04] Y. Bontemps, P. Y. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004.

[BZ83] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.

[CDO97] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[CF03] F. Coste and D. Fredouille. Unambiguous Automata Inference by Means of State-Merging Methods. In N. Lavrac, D. Gamberger, L. Todorovski, and H. Blockeel, editors, *Proceedings of the 14th European Conference on Machine Learning (ECML 2003) Cavtat-Dubrovnik, Croatia*, volume 2837 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CGV94] A. Castellanos, I. Galiano, and E. Vidal. Application of OSTIA to Machine Translation Tasks. In R. C. Carrasco and J. Oncina, editors, *Proceedings of the 2nd International Colloquium on Grammatical Inference and Applications (ICGI 1994), Alicante, Spain*, volume 862 of *Lecture Notes in Computer Science*, pages 93–105. Springer, 1994.

[Cho78] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[Chu57] A. Church. Applications of Recursive Arithmetic to the Problem of Circuit Synthesis. In *Summaries of talks presented at the Summer Institute for Symbolic Logic*, pages 3–50. 1957.

[CK06] F. Coste and G. Kerbellec. Learning Automata on Protein Sequences. In *Proceedings of the 7th Journées Ouvertes Biologie Informatique Mathématiques 2006 (JOBIM 2006), Bordeaux, France*, 2006.

[CP05] J. M. Champarnaud and T. Paranthoën. Random Generation of DFAs. *Theoretical Computer Science*, 330(2):221–235, 2005.

[CR79] E. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM*, 22(5):281–283, 1979.

[CSH03] I. Craggs, M. Sardis, and T. Heuillard. AGEDIS Case Studies: Model-based Testing in Industry. In *Proceedings of the 1st European Conference on Model Driven Software Engineering, Nuremberg, Germany*, pages 106–117, 2003.

[CW98] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[DH01] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DLD05] C. Damas, B. Lambeau, and P. Dupont. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[dlH97] C. de la Higuera. Characteristic Sets for Polynomial Grammatical Inference. *Machine Learning*, 27(2):125–138, 1997.

[dlH05] C. de la Higuera. A Bibliographical Study of Grammatical Inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[DLT02] F. Denis, A. Lemay, and A. Terlutte. Residual Finite State Automata. *Fundamenta Informaticae*, 51(4):339–368, 2002.

[DLT04] F. Denis, A. Lemay, and A. Terlutte. Learning Regular Languages Using RFSAs. *Theoretical Computer Science*, 313(2):267–294, 2004.

[Dup96a] P. Dupont. Incremental Regular Inference. In L. Miclet and C. de la Higuera, editors, *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences (ICGI 1996), Montpellier, France*, volume 1147 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 1996.

[Dup96b] P. Dupont. *Utilisation et Apprentissage de Modèles de Langage pour la Reconnaissance de la Parole Continue.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, 1996.

[Eas04] S. M. Easterbrook. Requirements Engineering. Unpublished manuscript at: `http://www.cs.toronto.edu/~sme/papers/2004/FoRE-chapter03-v8.pdf`, 2004.

[EGP07] E. Elkind, B. Genest, and D. Peled. Detecting Races in Ensembles of Message Sequence Charts. In O. Grumberg and M. Huth, editors, *Proceedings of 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), Braga, Portugal*, volume 4424 of *Lecture Notes in Computer Science*, pages 420–434. Springer, 2007.

[EGPQ06] E. Elkind, B. Genest, D. Peled, and H. Qu. Grey-box checking. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2006), Paris, France*, volume 4229 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2006.

[EMST03] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol Conformance for Logic-based Agents. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico*, pages 679–684. Morgan Kaufmann Publishers, 2003.

[EMST04] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Logic-Based Agent Communication Protocols. In F. Dignum, editor, *Workshop on Agent Communication Languages (ACL 2003), Melbourne, Australia*, volume 2922 of *Lecture Notes in Computer Science*, pages 91–107. Springer, 2004.

[FCC$^+$08] A. Farzan, Y. Chen, E. M. Clarke, Y. Tsay, and B. Wang. Extending Automated Compositional Verification to the Full Class of Omega-Regular Languages. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) Budapest, Hungary*, volume 4963 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008.

[FL79] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Science*, 18(2):194–211, 1979.

[Fu81] K. S. Fu. *Syntactic Pattern Recognition and Applications.* Prentice Hall, December 1981.

[FvBK$^+$91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[GdPAR08] P. García, M. Vazquez de Parga, G. Alvarez, and J. Ruiz. Learning Regular Languages Using Nondeterministic Finite Automata. In O. H. Ibarra and B. Ravikumar, editors, *Proceedings of the 13th International Conference on Implementation and Applications of Automata (CIAA 2008), San Francisco, California, USA*, volume 5148 of *Lecture Notes in Computer Science*, pages 92–101. Springer, 2008.

[Gen05] B. Genest. Compositional Message Sequence Charts (CMSCs) Are Better to Implement Than MSCs. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, UK*, volume 3440 of *Lecture Notes in Computer Science*, pages 429–444. Springer, 2005.

[GJM02] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 2nd edition, 2002.

[GKM06] B. Genest, D. Kuske, and A. Muscholl. A Kleene Theorem and Model Checking Algorithms for Existentially Bounded Communicating Automata. *Information and Computation*, 204(6):920–956, 2006.

[GKM07] B. Genest, D. Kuske, and A. Muscholl. On Communicating Automata with Bounded Channels. *Fundamenta Informaticae*, 80(1–3):147–167, 2007.

[GLP06] O. Grinchtein, M. Leucker, and N. Piterman. Inferring Network Invariants Automatically. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), Seattle, Washington, USA*, volume 4130 of *Lecture Notes in Artificial Itelligence*, pages 483–497. Springer, 2006.

[GMSZ06] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-State High-Level MSCs: Model-Checking and Realizability. *Journal on Computing and System Sciences*, 72(4):617–647, 2006.

[Gol67] E. Mark Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] E. Mark Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[GV90] P. Garcia and E. Vidal. Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, 1990.

[Har87] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.

[HMK$^+$05] J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni, and P. S. Thiagarajan. A Theory of Regular MSC Languages. *Information and Computation*, 202(1):1–38, 2005.

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., July 2006.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-specific Optimization in Automata Learning. In W. A. Hunt Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003), Boulder, USA*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[Hol94] G. J. Holzmann. The Theory and Practice of a Formal Method: NewCoRe. In *IFIP Congress (1)*, pages 35–44, 1994.

[HR04] G. Hamon and J. M. Rushby. An Operational Semantics for Stateflow. In M. Wermelinger and T. Margaria, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004), Barcelona, Spain*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2004.

[HV05] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. *Electronic Notes in Theoretical Computer Science*, 138(3):21–36, 2005.

[ITU96] ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96), 1996.

[ITU98] ITU-TS Recommendation Z.120 Annex B: Formal Semantics of Message Sequence Charts, 1998.

[ITU99] ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC99), 1999.

[ITU04] ITU-TS Recommendation Z.120 (04/04): Message Sequence Chart (MSC), 2004.

[JGr] JGraph Ltd. *JGraph* - `Java` *Graph Visualization and Layout.* Library available at: `http://www.jgraph.com/`.

[KGSB98] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES 1998), Schloss Eringerfeld, Germany*, volume 155 of *IFIP Conference Proceedings*, pages 61–72. Kluwer, 1998.

[Kle56] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Math. Studies 34*, pages 3–40. Princeton, New Jersey, 1956.

[Kof07] L. Kof. Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 2007), Delhi, India*, pages 121–130. IEEE Computer Society, 2007.

[Kof08] L. Kof. From Textual Scenarios to Message Sequence Charts: Inclusion of Condition Generation and Actor Extraction. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE 2008), Barcelona, Spain*, pages 331–332. IEEE Computer Society, 2008.

[KPP09] H. Kugler, C. Plock, and A. Pnueli. Controller Synthesis from LSC Requirements. In M. Chechik and M. Wirsing, editors, *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009), York, UK*, volume 5503 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2009.

[Leu07] M. Leucker. Learning Meets Verification. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006), Amsterdam, The Netherlands, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2007.

[`libalf`] `libalf`. Webpage: `http://libalf.informatik.rwth-aachen.de/`.

[Loh03] M. Lohrey. Realizability of High-level Message Sequence Charts: Closing the Gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.

[Lyn97] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1997.

[MCF03] S. J. Mellor, A. N. Clark, and T. Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003.

[Mor02] R. Morin. Recognizable Sets of Message Sequence Charts. In H. Alt and A. Ferreira, editors, *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2002), Antibes – Juan les Pins, France*, volume 2285 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 2002.

[MP95] O. Maler and A. Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995.

[MPO05] A. L. Martins, H. S. Pinto, and A. L. Oliveira. Using a More Powerful Teacher to Reduce the Number of Queries of the L* Algorithm in Practical Applications. In C. Bento, A. Cardoso, and G. Dias, editors, *Proceedings of the 12th Portuguese Conference on Progress in Artificial Intelligence (EPIA 2005), Covilhã, Portugal*, volume 3808 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2005.

[MRSL07] T. Margaria, H. Raffelt, B. Steffen, and M. Leucker. The LearnLib in FMICS-jETI. In *Proceedings of the 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), Auckland, New Zealand*, pages 340–352. IEEE Computer Society, 2007.

[MS01] E. Mäkinen and T. Systä. MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML. In *Proceedings of the 23th International Conference on Software Engineering (ICSE 2001), Toronto, Canada*, pages 15–24. IEEE Computer Society, 2001.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements Engineering: a Roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, pages 35–46. ACM, 2000.

[Nei03] O. Neise. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003.

[Ner58] A. Nerode. Linear Automata Transformation. *American Mathematical Society*, 9:541–544, 1958.

[Neu] H. Neukirchen. *MSC2000 Parser for Parsing MSC Files According to the ITU Z.120 Standard*. `helmut@hi.is (neukirchen@informatik.uni-goettingen.de)`.

[NP94] K. Najim and A. S. Poznyak. *Learning Automata: Theory and Applications*. Pergamon Press, Inc., Elmsford, New York, USA, 1994.

[OG92] J. Oncina and P. García. Inferring Regular Languages in Polynomial Updated Time. In *the 4th Spanish Symposium on Pattern Recognition and Image Analysis*, volume 1 of *Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, 1992.

[OS01] A. L. Oliveira and J. P. Marques Silva. Efficient Algorithms for the Inference of Minimum Size DFAs. *Machine Learning*, 44(1/2):93–119, 2001.

[Pel00] D. Peled. Specification and Verification of Message Sequence Charts. In *Formal Techniques for Distributed System Development, FORTE/PSTV 2000, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX), Pisa, Italy*, 2000.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977), Providence, Rhode Island, USA*, pages 46–57. IEEE Computer Society, 1977.

[Pre04] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2004.

[PVY02] D. Peled, M. Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[Ren98] M. A. Reniers. *Message Sequence Chart: Syntax and Semantics.* PhD thesis, Faculty of Mathematics and Computing, Eindhoven University of Technology, 1998.

[Roy87] W. Royce. Managing the Development of large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE 1987), Monterey, California, USA*, pages 328–338. IEEE Computer Society, 1987.

[RS06] H. Raffelt and B. Steffen. LearnLib: A Library for Automata Learning and Experimentation. In L. Baresi and R. Heckel, editors, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2006), Vienna, Austria*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer, 2006.

[RSB05] H. Raffelt, B. Steffen, and T. Berg. LearnLib: a Library for Automata Learning and Experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2005), Lisbon, Portugal*, pages 62–71. ACM, 2005.

[RV01] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 Volumes).* Elsevier and MIT Press, 2001.

[SC06] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. *IEEE Transactions on Software Engineering*, 32(8):587–607, 2006.

[Sha08] M. M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing.* PhD thesis, Laboratoire Informatique de Grenoble, 2008.

[Sip05] M. Sipser. *Introduction to the Theory of Computation.* Brooks Cole, 2nd edition, February 2005.

[`Smyle`] `Smyle`. Webpage: `http://www.smyle-tool.org/`.

[Som06] I. Sommerville. *Software Engineering.* Addison-Wesley Longman, Amsterdam, 8th edition, 2006.

[SP99] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components.* Object Technology Series. Addison-Wesley Longman, 1999.

[Tan02] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 2002.

[TB73] B. A. Trakhtenbrot and J. M. Barzdin. *Finite Automata: Behaviour and Synthesis.* North-Holland, 1973.

[TBD07] S. Trujillo, D. S. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA*, pages 44–53. IEEE Computer Society, 2007.

[Tho09] W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In L. de Alfaro, editor, *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009), York, UK*, volume 5504 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009.

[UBC07] S. Uchitel, G. Brunet, and M. Chechik. Behaviour Model Synthesis from Properties and Scenarios. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA*, pages 34–43. IEEE Computer Society, 2007.

[UKM03] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[usb] *USB*. USB 1.1 specification: `http://www.usb.org/developers/docs`.

[Vas73] M. P. Vasilevskii. Failure Diagnosis of Automata. *Cybernetics and Systems Analysis*, 9(4):653–665, 1973.

[VR81] Y. L. Varol and D. Rotem. An Algorithm to Generate all Topological Sorting Arrangements. *The Computer Journal*, 24(1):83–84, 1981.

[VSVA04] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to Verify Safety Properties. In J. Davies, W. Schulte, and M. Barnett, editors, *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004), Seattle, Washington, USA*, volume 3308 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.

[VSVA05] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using Language Inference to Verify Omega-Regular Properties. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, UK*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2005.

[WDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), Seattle, Washington, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006.

[WSK03] J. Whittle, J. Saboo, and R. Kwan. From Scenarios to Code: An Air Traffic Control Case Study. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA*, pages 490–497. IEEE Computer Society, 2003.

[Yok95] T. Yokomori. Learning Non-Deterministic Finite Automata from Queries and Counterexamples. *Machine Intelligence 13: Machine Intelligence and Inductive Learning*, pages 169–189, 1995.

List of Figures

1.1 Two MSCs from the Bluetooth® (version 3.0) specification 4

2.1 Finite-state acceptors for regular language L 12
2.2 Prefix tree acceptor for $S = \{aaa, ab, bb\}$ 15
2.3 Nondeterministic acceptors for regular language L 15
2.4 Minimal DFA for a regular language L' and corresponding canonical RUFA 18

3.1 Overview over classes of learning algorithms 22
3.2 Components of L^* and their interaction 36
3.3 An example of an L^* run 38
3.4 An example of an $\mathrm{L}^*_{\mathrm{col}}$ run 39

4.1 Three example tables of an NL^* run 48
4.2 Two tables $\mathcal{T}_1, \mathcal{T}_2$ and their corresponding NFA $\mathcal{R}_{\mathcal{T}_1}$ and $\mathcal{R}_{\mathcal{T}_2}$ 50
4.3 Inconsistency between table $\mathcal{T}$ and NFA $\mathcal{R}_{\mathcal{T}}$ 52
4.4 Minimal DFA for the (finite) regular language of Example 4.3.4 59
4.5 An NFA accepting regular language L_n 60
4.6 Minimal DFA $\mathcal{A}_2^*$ accepting language L_2 62
4.7 Canonical RFSA $\mathcal{R}_2$ accepting L_2 62
4.8 Final table $\mathcal{T}_4$ and corresponding canonical RUFA for language $\overline{L_2}$ 65
4.9 Number of states of minimal DFA and canonical RFSA 68
4.10 Number of membership queries for L^*, $\mathrm{L}^*_{\mathrm{col}}$, and NL^* 68
4.11 Number of equivalence queries for L^*, $\mathrm{L}^*_{\mathrm{col}}$, and NL^* 68
4.12 Number of states for $\mathrm{L}^*_{\mathrm{col}}$, NL^*, and UL^* 70
4.13 Number of membership queries for $\mathrm{L}^*_{\mathrm{col}}$, NL^*, and UL^* 71
4.14 Number of equivalence queries for $\mathrm{L}^*_{\mathrm{col}}$, NL^*, and UL^* 72

5.1 Minimal DFA for finite regular language L from Example 5.0.3 79

6.1 An MSC as a diagram and as a graph 84
6.2 A collection of message sequence charts 85
6.3 A collection of communicating finite-state machines 86
6.4 A deadlock configuration of a CFM 91
6.5 Some MSCs for a product language 93
6.6 Components of EXTENDED-L^* and their interactions 96
6.7 Schematic view of error cases for proof of Theorem 6.2.9 103

7.1 An MSC for explaining PDL 113
7.2 Sample MSCs for exemplifying the use of PDL 115
7.3 `SMA`: The `Smyle` Modeling Approach 119
7.4 The Waterfall lifecycle model 123
7.5 An evolutionary rapid prototyping lifecycle model 124

7.6 Extreme programming lifecycle model 125
7.7 Four positive initial input scenarios for `Smyle` 127
7.8 `Smyle`'s simulation window: 128
7.9 Patterns for (un)desired behavior 128
7.10 CFM for the alternating bit protocol 129

8.1 Architecture of the `libalf` learning library 135

9.1 Activity diagram for the *learning chain* `Smyle` is based upon 144
9.2 `Smyle`'s architecture: learning overview 145
9.3 `Smyle`'s graphical user interface 146
9.4 Choosing a learning setup in `Smyle` 147
9.5 Choosing and classifying a set of input MSCs for `Smyle` 148
9.6 MSC editor integrated into `Smyle` 149
9.7 Specifying some sample PDL formulae in `Smyle` 150
9.8 Simulating an intermediate model in `Smyle` 151
9.9 Simulating a final model in `Smyle` 152
9.10 `Smyle` input and output for a `USB` protocol 153
9.11 `Smyle` input and output for a negotiation protocol 155
9.12 `Smyle` input and output for the alternating bit protocol 156
9.13 `Smyle` input and output for the leader election protocol 158
9.14 `Smyle`'s architecture: component overview 162
9.15 Hierarchical learning example featuring two services 165
9.16 The model-based testing approach in general 166
9.17 The model-based testing approach using `Smyle` and `Style` 166
9.18 Test case generation using `Style` 167
9.19 Some test cases derived by `Style` 168

B.1 DFA: An example where NL* needs more membership queries than L* . . . 174
B.2 RFSA: An example where NL* needs more membership queries than L* . . 174
B.3 Acceptors: An example where NL* needs more equivalence queries than L* 177
B.4 DFA: An example where the intermediate hypothesis is not an RFSA . . . 179
B.5 NFA: An example where the intermediate hypothesis is not an RFSA . . . 179
B.6 DFA, NFA: An example for non-termination of algorithm from Section 4.5 181
B.7 DFA: An example for a non-increasing number of states 183
B.8 RFSA: An example for a non-increasing number of states 183
B.9 DFA: An example for a decreasing number of states 185
B.10 RFSA: An example for a decreasing number of states 185
B.11 DFA: NL* needs more equivalence queries than minimal DFA has states . . 187
B.12 RFSA: NL* needs more equivalence queries than minimal DFA has states . 187

List of Tables

3.1 Biermann: (passive) offline learning algorithm for inferring minimal DFA . 26
3.2 RPNI: (passive) offline learning algorithm for inferring (minimal) DFA . . . 28
3.3 DeLeTe2: (passive) offline learning algorithm for inferring (canonical) RFSA 31
3.4 L^*: active online algorithm for inferring minimal DFA 35
3.5 Function for updating table function in L^* 35
3.6 L^*_{col}: a variant of Angluin's algorithm L^* 40
3.7 Function for updating table function in L^*_{col} 40
3.8 LA: active online learning algorithm for inferring RFSA 41
3.9 An overview over the characteristics of the presented learning algorithms . 44

4.1 NL^*: active online learning algorithm for inferring canonical RFSA 54
4.2 An NL^* run and its corresponding measures 58
4.3 Learning regular language L_2 employing L^* 60
4.4 Learning regular language L_2 employing NL^* 61
4.5 Comparing NL^* to L^* and L^*_{col} (2-letter alphabet) 69
4.6 Comparing NL^* to L^* and L^*_{col} (3-letter alphabet) 69

5.1 cclL^*: Extension of L^* by congruence-closed language learning 77
5.2 Function for updating table function in cclL^* 77
5.3 Compression example for congruence-closed language learning 79

6.1 EXTENDED-L^*: The extension of Angluin's algorithm for learning CFMs . 98
6.2 Function for updating the table in EXTENDED-L^* 99
6.3 An overview over the query complexity of learning CFMs 108

8.1 A pseudocode testcase for Biermann's algorithm 138
8.2 `Java` code for simulating L^* with human *Teacher* using the dispatcher . . . 140
8.3 `C++` code for simulating L^* with human *Teacher* using `libalf` directly . . . 141
8.4 `Java` code for simulating L^* with human *Teacher* using JNI 142

9.1 `Smyle`: statistics on the inferred protocols 159

B.1 DFA: An example where NL^* needs more membership queries than L^* . . . 174
B.2 RFSA: An example where NL^* needs more membership queries than L^* . . 175
B.3 DFA: An example where NL^* needs more equivalence queries than L^* . . . 177
B.4 RFSA: An example where NL^* needs more equivalence queries than L^* . . 178
B.5 An example where the intermediate hypothesis is not an RFSA 180
B.6 An example for non-termination of algorithm from Section 4.5 181
B.7 An example of an NL^* run where the number of states does not increase . 184
B.8 An example of an NL^* run where the number of states even decreases . . . 186
B.9 NL^* might need more equivalence queries than minimal DFA has states . . 189

C.1 Algorithm for checking local formulas 191

C.2 Algorithm for checking forward path expressions 192
C.3 Algorithm for checking backward path expressions 193

Index

Symbols

Biermann . . . 25
cclL^* . . . 77
DeLeTe2 . . . 30
Extended-L^* . . . 99
L^* . . . 34
L^*_{col} . . . 37
LA . . . 41
NL^* . . . 55
RPNI . . . 27
UL^* . . . 65

A

Action . . . 82
Agile models . . . 125
  extreme programming . . . 126
  user stories . . . 126
Alphabet . . . 10
Alternating bit protocol (ABP) . . . 126, 155
Assistant . . . 145
Automata equivalence . . . 12

B

Biermann . . . 25
Bluetooth® . . . 4

C

Canonical residual alternating finite-state automaton (canonical RAFA) . . . 73
cclL^* . . . 77
Channel . . . 83
Closed table . . . 36
Communicating finite-state machine (CFM) . . . 5, 87
  accepting run . . . 88
  configuration . . . 87
  deadlock-free . . . 90
  deterministic . . . 90
  existentially B-bounded . . . 92
  final configuration . . . 87
  global transition relation . . . 87
  initial configuration . . . 87
  run . . . 88
  size . . . 87
  universally B-bounded . . . 92
  universally bounded . . . 92
  weak . . . 92
Composed residual language
  $\cap$-composed (RUFA) . . . 18
  composed (RFSA) . . . 16
Concatenation of words . . . 10
Configuration of a CFM . . . 87
  final . . . 87
  initial . . . 87
Congruence relation . . . 10
  Nerode right congruence . . . 10
  right congruence . . . 10
Congruence-closed language learning . . . 75
Consistent . . . 25, 27
Consistent table . . . 36
Constructive decidability . . . 96
Control message . . . 87
Covering relation . . . 47
  strict . . . 64

D

DeLeTe2 . . . 30
Deterministic finite-state automaton (DFA) . . . 12
Diamond property . . . 102

E

EqClosure($\mathcal{D}$) . . . 96
Equivalence class . . . 9
Equivalence query . . . 34 f., 143
Equivalence relation . . . 9

F

Finite index . . . 11

Finite-state automaton
  canonical RFSA 16
  canonical RUFA 18
  DFA 12
  NFA 12
  RFSA 16
  RUFA 17
  UFSA 17

I

Identification in the limit 23
  from polynomial time/data 24
INCLUSION($\mathcal{D}$) 96
Index of a language 11
INFCLOSURE($\mathcal{D}, \vdash$) 96
Intersection operator 64

J

Join operator 47

K

Kernel .... 29 f.

L

L* 34
L^*_{col} 37
`LA` 41
Language
  CFM 88
  language of a state 12
  NFA, DFA 12
  word language 10
Learnability of CFMs 98
Learner 145
Learning algorithm
  active 22
  offline 22, 24
  online 22, 33
  passive 22
Learning chain 144
Learning setup 95
Length-lexicographical order 10
`libalf` .... 134 – 139
Linearization 84

M

Membership query .... 33 ff., 143
Message Passing Automaton (cf. CFM) . 82
Message Sequence Chart 3, 84
  existentially B-bounded 91
  linearization 84
  universally B-bounded 91
Minimal adequate teacher (`MAT`) 34
Minimal automaton 12
  NFA, DFA 13
MSC 3, 84

N

Nerode right congruence .... 10 f.
NFA of a table 49
NL* 55
Nondeterministic finite-state automaton (NFA) 12

O

Oracle 34, 150
Order
  length-lexicographical 10

P

PDL 111, 120
  local formula 112
  path expression 112
  PDL (global) formula 112
  semantics 112
  syntax 112
Polynomial learner 24
Power set 9
Prefix automaton 14
Prefix set 10
Prefix tree acceptor 14
Prime residual 16
Prime residual language 16
  ∩-prime (RUFA) 18
  prime (RFSA) 16
Prime state 16
Productive state of a DFA 101
Protocols
  alternating bit 126, 155
  leader election 157
  negotiation 154
  USB 1.1 154

Q

Query

  - equivalence 34
  - membership 33

R

- Rapid prototyping 124
  - evolutionary prototyping 124
  - throw-away-prototyping 124
- Regular expression 11
- Regular language 11
- Relation 9
  - binary 9
- Representation class 24
- Residual alternating finite-state automaton (RAFA) 73
  - canonical 73
- Residual finite-state automaton (RFSA) 16
  - canonical 16
- Residual language 11
  - ∩-composed 18
  - ∩-prime 18
  - composed 16
  - prime 16
- Residual universal finite-state automaton (RUFA) 17
  - canonical 18
- RFSA-closed 48
- RFSA-consistent 49
  - strong 62
  - weak 63
- RPNI 27
- RUFA-closed 64
- RUFA-consistent 64

S

- Sample 25, 27
  - complete 29
  - complete for inclusion relations 30
- Scenarios 117
- Shortest prefix 29 f.
- Size of an automaton 12
- `SMA` 111 – 129
- `Smyle` 143 – 164
- `Smyle` Modeling Approach 111 – 129
- Spiral model 124
- State 12
  - final 12
  - initial 12
  - prime 16
  - productive 101
- Suffix-set 10
- Synchronization message 87

T

- Table
  - closed 36
  - consistent 36
  - consistent with table 52
  - RFSA-closed 48
  - RFSA-consistent 49
  - RUFA-closed 64
  - RUFA-consistent 64
- Table row
  - ⊓-composed (UFSA) 64
  - ⊓-covered (UFSA) 64
  - ⊓-prime (UFSA) 64
  - composed (RFSA) 47
  - covered (RFSA) 47
  - prime (RFSA) 47
  - strictly covered (RFSA) 47
- *Teacher* 34, 149
- Transition function 12

U

- UFSA of a table 65
- UL* 65
- Universal finite-state automaton (UFSA) 17
- User query 97, 143

V

- V-model 123

W

- Waterfall model 116, 123
- Word 10
  - B-bounded 91
  - proper 88
  - well-formed 88

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises “Features”
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19	Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04	Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Curriculum Vitae

Name	Carsten Kern
Geburtsdatum	8. Mai 1979
Geburtsort	Aachen

Bildungsgang

1989–1998	Stiftisches Gymnasium Düren Abschluss: Allgemeine Hochschulreife
1998–1999	Zivildienst im St. Augustinus Krankenhaus Lendersdorf
1999–2005	Studium der Informatik an der RWTH Aachen Abschluss: Diplom
2005–2009	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik 2 (Prof. Dr. Ir. Joost-Pieter Katoen), RWTH Aachen