Automata Learning for Lightweight Formal Methods

Tiago de Faria Carvalho Morais Ferreira

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

I, Tiago de Faria Carvalho Morais Ferreira, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Lightweight formal methods bring the guarantees of formal verification within reach of modern software development by trading blanket completeness for pragmatic partiality. The resulting techniques are more amenable to automation, reward incremental rather than upfront formalization effort, and remain accessible to practitioners without verification expertise.

Automata learning is a particularly promising enabler of this vision: by automatically inferring formal behavioral models from system interactions, it removes the upfront cost of manual modeling and opens model-based analysis to systems that would otherwise resist formalization.

This thesis contributes new automata learning algorithms and tools that bring automata learning closer to the demands of modern software development. We introduce PROGNOSIS, a push-button, closed-box framework for analyzing network protocol implementations. Its configurable abstraction levels let practitioners trade detail for cost without verification expertise, and its bug-finding workflow has uncovered specification ambiguities and security-critical defects in production QUIC protocol stacks. We then develop IKV, the first tree-based adaptive learning algorithm, which prunes and reuses prior models when systems evolve, outperforming the prior state of the art. We generalize this approach with CεAL, an adaptive learning framework built around the observation tree that replaces the classical Minimally Adequate Teacher. CεAL enables existing algorithms to learn under noise, partial observability, and change, achieving a 95.5% success rate against MAT's 79.5%. Together, IKV and CεAL improve the incrementality and robustness of learning itself, reducing the human effort needed to keep learned models current. Finally,

we present SWAL, an SMT-based active learning algorithm that casts learning as constraint solving over a state-local characterization of weighted automata. SWAL extends these benefits to the quantitative domain, inferring minimal models over arbitrary semirings and capturing costs, latencies, and bottleneck behaviors that lie beyond the expressive reach of Boolean models.

Impact Statement

The field of Lightweight Formal Methods [WJJ96] aims to bring the strong correctness guarantees of formal verification into practical software development settings, where techniques must be flexible, incremental, and applicable without verification expertise. This thesis advances that objective through contributions to Automata Learning, which automatically constructs formal models of a system's behavior from its interactions. By eliminating the need for manual modeling, these techniques substantially reduce upfront cost and mitigate human error. This, in turn, enables bug finding and model-based analysis in real-world software systems.

The tools developed in this thesis have had direct industrial impact. PROGNOSIS (Chapter 3) identified security-critical defects in widely deployed network protocol implementations. It discovered a Denial-of-Service vulnerability in Facebook's mvfst, a core component underlying Facebook, Instagram, and WhatsApp services. It also uncovered bugs in Google's QUIC implementation and in the Google Chrome browser. Furthermore, PROGNOSIS was applied within a US government project to evaluate the suitability of a network stack implementation for safety-critical use. These results demonstrate the feasibility of applying model-based formal techniques to production systems at global scale.

The need to operate in evolving and imperfect real-world environments motivated contributions to adaptive automata learning. IKV (Chapter 4) enables learning-based tools to incrementally update models when systems change, avoiding costly relearning from scratch. CεAL (Chapter 5) generalizes this idea by introducing a conflict-aware learning framework in which classical algorithms can operate robustly under noisy, inconsistent, or evolving observations. Together, these contributions

strengthen the applicability of automata learning in dynamic deployment settings.

The SWAL algorithm (Chapter 6) advances the state of the art in learning in the quantitative domain. While earlier tools like PROGNOSIS could capture qualitative control-flow structure, they struggled with quantities induced by execution, such as latency or routing cost. SWAL advances practical learning of such models and leverages a direct theoretical characterization of quantitative behavior. This has led to progress on longstanding problems, including exact learning of minimal NFAs and improved performance in quantitative model inference.

Academically, PROGNOSIS helped broaden the scope of automata learning beyond its initial emphasis on network protocols. Subsequent work has applied learning-based techniques to diverse domains, including memory vulnerability discovery [Nas+22], network routing and transport behavior [Moe+25], specification mining [Rot+25], as well as network fuzzing and analysis [Jia+24; ABA23]. More generally, it helped motivate research on richer model classes, such as timed Mealy machines [VEB23; Bru+26] for capturing temporal behavior and automata with registers [Die+24] for handling data-dependent state. The adaptive learning contributions likewise inspired further algorithmic developments, including the $AL^\sharp$ algorithm [KJR25] used in device fingerprinting techniques [Kru+26].

The body of work presented in this thesis has been published in international peer-reviewed venues. It has supported fruitful research visits in the United States, Germany, and the Netherlands, where it allowed for engaging discussions and collaborations with experts in the field.

Acknowledgements

Throughout my academic career I have had the privilege of truly standing on the shoulders of giants. I am eternally grateful to my supervisor Alexandra Silva, who has kindly mentored me for the last ten years. When I was in high school, during a summer break with nothing to do, Alexandra very kindly agreed to let me hang out with her group, and learn about what research is.

The first paper I ever "read" was entirely beyond my comprehension, but through Alexandra's guidance and enthusiasm, I was shown an environment where what seemed impossible to me was being achieved as a matter of routine, and Alexandra made me feel truly welcome and already a part of it.

Over the years, Alexandra's help reading papers turned into help running experiments, writing papers, giving talks, encouraging me to get my Bachelor's, writing more papers, supervising a student, starting a PhD program, writing more papers, visiting many people, workshops, conferences, and countries, doing an internship in industry, writing more papers, ... and getting a PhD! Interleaved with these professional milestones, Alexandra helped me grow as a person. It builds your faith in humanity to see someone who operates at such an elite academic level also be a genuinely great person to everyone they interact with. I will miss our one-to-ones, which always started with the more important things: how is life, how is the family, how is your health, your rest, your work-life balance? A PhD is quite a long journey, and it is important to not lose track of the important things in life.

I will never be able to repay the incredible impact Alexandra has had on my life, but I hope I can pay it forward, thanks to the example she has set of how to be a truly present, caring, and impactful mentor. Alexandra, a truly heartfelt *thank you*.

I would like to thank my examiners, Clemens and Mohammad, for the great care and attention to detail they took in evaluating this thesis, which led to a very pleasant thesis defense. The hours flew by, and I am honored for the opportunity to discuss matters so dear to me with such experts in the field. Thank you also for your very detailed review and suggestions, which made this thesis significantly better.

Thank you to all my coauthors over the years: Alexandra, Loris, Matteo, Harrison, Gerco, Léo, Raquel, Mark (Moeller), Thomas, Nate, Noga, Hila, Mark (Silberstein), Kevin, Emma, Oliver, Robin. It was a blast working with every one of you, on so many projects! Thank you in particular to the senior coauthors that mentored me along the way: Alexandra, Loris, Matteo, Léo, Kevin, and Nate.

Thank you to my first student, Raquel, for being so patient with me as I learned to be a mentor myself. I am very proud of all you have achieved and will continue to achieve, and thankful for all that you taught me while we worked together!

Thank you to Mike Dodds, Mike Beynon, Michal Podhradsky, and all the great folks at Galois, for hosting me for an internship, and allowing me to apply my research to the real-world! I had a blast working in Portland those few months, and I marvel at the incredible culture of support in scientific discovery that you've managed to foster at Galois. It is truly *the Goldilocks zone of academia and industry*.

Thank you to Nate Foster, Martin Leucker, Jurriaan Rot, and Frits Vaandrager, for hosting me on fruitful and engaging visits over the years. Many ideas were born in these visits, and led to great papers and collaborations! Thank you to the organizers of VMW'19, CMMRS'22, PLMW'23 and '25, and FLoC'26 for putting together very helpful mentorship workshops, and the organizers of the Bellairs Workshop on Quantitative Reasoning for their invitation to such an inspiring meeting.

Thank you to Robin Hirsch, my undergraduate personal tutor, for cutting me a lot of slack with my sporadic attendance. This allowed me to balance my personal life, student life, and research life. Thank you also to Earl Barr, for engaging conversations on the future of formal verification while we marked piles of exams!

Thank you to Steve Hailes, Rae Harbird, Cathy Holloway, and Elaine Pimentel, who allowed me to get involved and help in the School's Outreach program over the

years. It was great to develop activities that get to the core of computational and algorithmic thinking, that can be taught to anyone, anywhere! The program reached over three thousand students in over fifty-five schools. In 2025, 100% of students in the A-COMPS mentoring program secured a university place. The small contribution I made to this program remains the most important one of my professional career.

On a more personal level, I have been supported by so many wonderful people in this journey.

Thank you to professores Rui and João of CTEM Academy for introducing me to robotics, and teaching me how to use an Arduino. This was my introduction to computer science. Without them, I might never have found my way into computing. Our work in Brazil is still my biggest claim to fame!

Thank you to Mrs Hodson for showing me the wonderful beauty in mathematics. There was never anything I couldn't ask "why" about, that did not get a wise answer! Thank you also to Mrs Sawa, who was very supportive of me and my family during tough times in the pandemic.

Thank you to Joshila, Phil, Kevin, and all the great people at WCS, for your confidence in me, allowing me to restructure your internal ops infrastructure and learn so much along the way. More importantly, thank you for your support, patience, and kindness when I was going through a rough point with my health. I felt looked after way beyond what an employer and colleagues would usually settle for.

Thank you to Steffen Smolka for teaching me what a monad is, and to Ben van Enckevort for showing me what a monad does. Thank you to Tobias Kappé for making sure I didn't stop at monads, by explaining what a coalgebra is!

Thank you to nurse Becky, and nurses Holly, Romanie, Ella, and Anna of the RSCH MDU for keeping me healthy for almost ten years now. They have been a constant source of care, kindness, and reassurance, and almost always my *de facto* first point of contact whenever something went wrong. They work tirelessly, often with far too little recognition and far too few resources, quietly keeping the NHS moving and shielding patients from pressures they should never have to bear.

Thank you to the GS 3rd floor crowd: Gerco, Tobias, Robin, Fred, Steffan,

Benjamin, and Matteo, who welcomed me into the lab, supported me through my first steps in research, and even helped me with my school homework!

Thank you to the subsequent GS basement crowd: Fred, Robin, Sam, Mateo, Wojtek, Leo (Henry and Lobski), Katya, Ralph, Rafa, Louis, Jas, Oliver, Kevin, Cheng, Linpeng, Luigi, Stefan, and Will, for your friendship, guidance, and all the fun *state mandates*. I have so many fond memories of our trips and time together, and I will miss you all dearly.

Thank you to Todd and Tobias, who hosted me for my first visit to Ithaca / Cornell, for showing me around town, and taking me to cool music gigs and kayaking in the Finger Lakes!

Thank you to Mark and Erin for their friendship, for hosting me (twice!) in their home in Ithaca, and for making my visits warm and memorable. Thank you for taking me on hikes, driving us west to see a total eclipse, and for our discussions of baseball, golf, tennis, and basketball. Maybe one day we will hit the NCAA bracket jackpot! Until then, I maintain that the most orange team will eventually win.

Thank you to my childhood friends from Portugal, Miguel, Pedro Barbosa, Pedro Antunes, Raul, João, and Tiago. No matter how far I am from you, it is like nothing has changed when we are back together.

Thank you to the *gangue dos reis*: Carlitos, Paulinha, Joao, Beatriz, Francisco, Susana, Miguel, Diogo, Lucas, Rita, Carolina, and Margarida, for their warm support over the years, and helping me celebrate this achievement *à grande e à francesa*.

Thank you to the Austins, Geoff and Antonia, for always welcoming me into their home, looking after me, and pampering me beyond reason, and for creating such a wonderful environment in which so much of this thesis was written.

Obrigado a todos os Farias, os Carvalhos, os Morais, os Ferreiras, e mais: Avó Bel, Avô Fernando, Tatá, Terry, Avó Xinha, Avô Malhão, Malú, Senhor Alves, Dona Lurdes, Kokas, Fernanda, Arthur, Juliana, Palmira, Margarida, Paulinha, Madalena, Eliane, Dona Fátima, Prudencio, Chipolata, Maurice, João, Ana, Licas, Queta, Maju, Xavi, Matilde, e Filipe! Tenho uma sorte enorme por ter uma família tão grande, carinhosa e generosa. Tenho saudades de todos vocês e, sempre que volto a casa, a

Portugal, mostram-me o quanto também tiveram saudades minhas.

Thank you to my *amigos do peito*, Varghese, Danny, and Darren, who kept me sane and happy through the many years of research and thinning hair. I am glad that even though this chapter comes to an end for me, we will still have our movie and game nights, and pool outings. I could always count on you, and hope you know you can count on me too.

Thank you to my siblings. Thank you, Sofia, for being a bastion of creativity and academic excellence. I look up to you, and your example has given me strength through many challenging times. Thank you, Vasco, for looking after me, cooking incredible meals, and being patient and warm, even when I left the place a mess during stressful deadlines. Thank you, Sarinha, Pocoyo, my *craque dos jogos*. I am proud of everything you have achieved, and proud to see you launch yourself into computing. Never let anyone tell you what you can and cannot do. Thank you, Catarina, Piolha, for being the dimple in the smile of our family. You are young enough that I have had the privilege of watching you grow into the wonderful person you are. Believe in yourself, and stay true to what makes you happy.

Thank you to my parents, to whom I dedicate this thesis, for showing me, through your actions, thoughts, and words, how deeply you love and support me. You have sacrificed much to put us first and to make sure that my siblings and I had everything we needed, and more. It fills my heart knowing how proud I make you, and I hope you know how proud I have been of both of you, and always will be.

Thank you to my wonderful partner Jess, for all your love and unwavering support before, throughout, and beyond this journey. You have helped me through every low, and celebrated every high. I could not have done this without you, and I am so happy and fortunate to continue into this new phase of life by your side.

Contents

1 Introduction **19**

1.1 Formal Methods 19

1.2 Lightweight Formal Methods 21

1.3 Automata Learning 23

1.4 Thesis Overview 26

1.4.1 Learning Network Protocol Implementations (Chapter 3) . . 26

1.4.2 Learning Evolving Systems (Chapter 4) 28

1.4.3 Learning Conflicting Systems (Chapter 5) 29

1.4.4 Learning Quantitative Systems (Chapter 6) 31

1.4.5 Other Contributions 33

2 Preliminaries **35**

2.1 Formal Languages 35

2.2 Automata Theory 36

2.2.1 Transducers 40

2.3 Automata Learning 42

2.3.1 The Minimally Adequate Teacher 42

2.3.2 The Observation Table 43

2.3.3 The $L^\star$ Algorithm 46

2.3.4 Other Learning Algorithms 47

3 Closed-Box Analysis of Network Protocol Implementations **50**

3.1 Architecture 53

3.1.1 System Under Learning (SUL) . . . 53
3.1.2 Learning Module . . . 54
3.1.3 Analysis Module . . . 54
3.2 System Under Learning (SUL) . . . 55
3.2.1 Implementation . . . 55
3.2.2 Adapter . . . 57
3.3 Learning Module . . . 64
3.3.1 Learner Interface . . . 64
3.3.2 Synthesizing Rich Models . . . 65
3.4 Analysis Module . . . 70
3.4.1 Nondeterminism Check . . . 70
3.4.2 Learned Model Analysis . . . 71
3.5 Results . . . 71
3.5.1 Learning a TCP Implementation . . . 72
3.5.2 Learning QUIC Implementations . . . 74
3.5.3 Background . . . 74
3.5.4 Learning Models of QUIC Implementations . . . 75
3.5.5 Issue 1: RFC Imprecision . . . 77
3.5.6 Issue 2: Nondeterminism in Connection Closure . . . 78
3.5.7 Issue 3: Inconsistent Port on RETRY in QUIC-Tracker . . . 79
3.5.8 Issue 4: Stream Data Blocked Bug in Google QUIC . . . 80
3.6 Related Work . . . 81
3.7 Discussion . . . 82

4 Tree-Based Incremental Automata Learning **84**

4.1 Learning with a Classification Tree . . . 85
4.1.1 Constructing Hypothesis Automata . . . 88
4.1.2 Refining From Counterexamples . . . 90
4.1.3 The Kearns–Vazirani Algorithm . . . 93
4.2 Learning Evolving Systems Incrementally . . . 94
4.2.1 Correctness and Termination . . . 97

4.3 Experimental Evaluation 98
4.3.1 Implementation 98
4.3.2 Research Questions and Methodology 99
4.3.3 Results and Analysis 105
4.4 Related Work 108
4.5 Discussion 109

5 Conflict-Aware Active Automata Learning 110
5.1 Conflicts in Automata Learning 112
5.1.1 Observation Noise 113
5.1.2 System Evolutions 115
5.2 The Framework 117
5.2.1 Overview 117
5.2.2 The Reviser 119
5.2.3 Interface Implementation 126
5.2.4 Learning Lifecycle 127
5.3 Experimental Evaluation 129
5.3.1 Research Questions and Methodology 130
5.3.2 Results and Analysis 132
5.4 Related Work 137
5.5 Discussion 138

6 SMT-Based Active Learning of Weighted Automata 140
6.1 Semirings and Weighted Automata 142
6.2 Learning Algorithm 144
6.2.1 The Teacher 144
6.2.2 The Main Loop of the Algorithm 145
6.3 Computing n-State O-Correct WFAs 148
6.3.1 A Naive Baseline Implementation 149
6.3.2 An Optimized Implementation via State Languages 150
6.4 Deciding Equation Systems via SMT Solving 156

6.4.1 The Tropical Semiring . . . 156
6.4.2 The Bounded Tropical Semirings . . . 156
6.4.3 The Bottleneck Semiring . . . 157
6.5 Experimental Evaluation . . . 158
6.5.1 Implementation . . . 158
6.5.2 Research Questions and Methodology . . . 159
6.5.3 RQ1: Ablation Study . . . 160
6.5.4 RQ2: Comparison with NAIVE Algorithm . . . 162
6.5.5 RQ3: Comparison with WL* Algorithm . . . 163
6.6 Related Work . . . 165
6.7 Discussion . . . 166

7 Conclusion and Future Work 168

Appendices 172

A PROGNOSIS Models 172
A.1 Learned Mealy Machines . . . 172
A.1.1 TCP Implementation . . . 172
A.1.2 Google's QUIC . . . 174
A.1.3 Cloudflare's Quiche . . . 176
A.2 Synthesized Extended Mealy Machines . . . 178
A.2.1 Google's QUIC – `Maximum Stream Data` Field . . . 178

B CℰAL Results 180

List of Figures

1.1 Language information cover of observation table vs classification tree. 28

1.2 Pruning a previous tree (a) learned for language $\mathcal{L}$ into a tree (b) containing only information consistent with the new (closed-box) target language $\mathcal{L}'$. 29

1.3 Two observation trees and their languages. 30

1.4 A 2-state WFA over the Tropical semiring $\mathbb{S} \triangleq \langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0 \rangle$. 31

3.1 PROGNOSIS' modular architecture. 53

3.2 System Under Learning components. 55

3.3 TCP handshake: protocol, learned model, and synthesized model. . 57

3.4 Extended machine with unknown terms. 67

3.5 Synthesized machine over registers *r*, *pi*, and *pr*. 68

4.1 Mutation Benchmark: Average learning progress graphs. 105

4.2 Feature Benchmark: Average learning progress graphs. 105

4.3 Ratio of number of queries since 10K needed to fully learn the t_{10K} target. 107

5.1 Simplified view of the CƐAL framework. See Figures 5.3 and 5.4 for details. 117

5.2 The Reviser's interfaces. 118

5.3 Implementation of Membership Queries (*mq*) in the Reviser. 126

5.4 Implementation of Equivalence Queries (*eq*) in the Reviser. 127

5.5 INPUT noise at 0.01. 133

5.6 INPUT noise at 0.05. 133

5.7 INPUT noise at 0.1. 133
5.8 OUTPUT noise at 0.01. 134
5.9 OUTPUT noise at 0.05. 134
5.10 OUTPUT noise at 0.1. 134

6.1 Equation system $\Phi(O,n)$ characterizing the existence of n-state O-correct WFAs. Here $(f_i^w)_{i\in n, w\in \mathsf{suffixes}(O)}$, $(\delta_{i,j}^a)_{a\in\Sigma,(i,j)\in n\times n}$, and $(\iota_i)_{i\in n}$ are $\mathbb{S}$-valued variables. 154
6.2 Effect on learner time (seconds in log-log) of incremental SMT solving (left), and bit vector (QF-BV) vs linear integer arithmetic (QF-LIA) encoding for bounded tropical semirings (right). TO/OOM count provided per axis. 161
6.3 Comparison of NAIVE vs SWAL on average total run-time (seconds in log-log, left), and number of membership queries (count, right), over number of states. 162
6.4 Comparison of WL* vs SWAL on total run-time (seconds in log-log, top left), number of membership queries (count in log-log, top right), and number of states learned (log-log, bottom). TO/OOM count provided per axis (top left). 163

List of Tables

3.1 Summary of PROGNOSIS results on QUIC implementations. 77

Chapter 1

Introduction

Computer systems are ubiquitous in modern society and our day-to-day lives. We have for decades now entrusted computers to perform safety-critical actions, such as auto-piloting our planes, protecting our money, securely exchanging it across the world, and even keeping our hearts beating when our bodies start to fail us. This trust was not easily given, but rather earned: Through government bodies we developed tight regulations on the development of such devices, and set high bars of what it would take for society to accept the risk of these life-changing innovations.

Often, one such requirement is that rigorous assurances be given as to the correctness of the device or software being developed. While methodologies like testing can help sanity check specific expectations of the behavior of software, safety-critical systems often require a more *complete* assurance: that given some notion of the correct behavior of a system and its implementation, one can ensure that the system fully meets the correctness requirements.

The central purpose of this thesis is to contribute new techniques for providing strong guarantees on the modeling and correctness of computer systems, even in the setting of fast changing, continuous software development.

1.1 Formal Methods

Formal Methods are a collection of techniques for *specifying* the intended behavior of systems, and *verifying* that an implementation meets such requirements. The methods are *formal* in the sense that if we *formalize* the ideal behavior of the system

as a mathematical property, then we can devise provably sound techniques for determining whether it indeed holds in the target system. In doing so, we establish a rigorous framework for producing mathematical proofs of system correctness. By grounding both specification and behavior in formal reasoning, we reduce the margin for human error in what the system is intended to do and what it demonstrably does.

Formal methods have their roots in program semantics and their proof systems of the late 1960s and early 1970s [Flo67; Dij75; Lav75; Coo78; Gor79; Lon+79]. Works such as Floyd-Hoare Logic [Hoa69] and Scott's Denotational Semantics [Sco70] represent some of the first attempts at formalizing what it means for a program to do something, or meet a certain property. These foundations allowed the field of *model-based* formal specification languages to flourish, with the development of VDM and Z in the late 1970s [BJ78; ASM80]. The working principle is to specify the correct behavior of a system by first *modeling the state* of a program in terms of a mathematical object, and then specifying the properties that the model must meet upon running a certain subroutine. In this way, the specification language does not concern itself with how the subroutine is to be implemented, but rather the effect it will have on the state once it has finished.

To put these formal specifications into action, in the 1980s we saw the development of *model checking* [CES86]: the idea that determining whether a system meets a formal specification amounts to proving that certain properties are satisfied in a mathematical structure modeling the system's behavior. The model in question is often a Finite State Machine (FSM), representing how the state of a system changes as the program progresses. The property being checked is classically a statement in a temporal logic, such as "Eventually the variable x is equal to $y+1$.". The model checker (through graph algorithms or model-specific methods) then proves that the property holds for the model, or provides a trace witnessing its violation.

Model-based formal methods, such as model checking, have enabled establishing the correctness of a wide range of safety-critical systems. They have been used to verify aerospace and avionics systems [Sou+09; Moy+13], transportation systems such as railway signaling [GCR94; Ter+19], health systems such as pace-

makers [Jia+12] and infusion pumps [HCM15], and many more systems we rely on to lead our lives [Ter+25].

We are, however, living through a shift in paradigm: Our lives are increasingly affected by technology produced not by government agencies or tightly regulated mega-corporations, but by startups or even hobbyists developing software that is relied upon globally. While formal methods provide an undeniable value, they also come at a great cost. Not only is verifying a real-world piece of software time consuming, it also usually requires expert knowledge of formal methods and of the software at hand. Together, these imply a great financial cost which small companies can rarely justify, and open source projects cannot afford.

1.2 Lightweight Formal Methods

To alleviate this disconnect, in the mid-1990s Wing et al. [WJJ96] introduced the concept of *Lightweight* Formal Methods. The idea is that by being pragmatic and reducing the completeness requirements of certain formal methods, one can vastly reduce their cost and make them viable for fast-moving industries. The authors highlight the need for *partiality*: as long as one retains soundness of the results, the specifics of the application should determine the level of cover and detail provided by the assurances, rather than aiming for a blanket proof of full system correctness.

In practice, this is achieved through deliberate partiality in language, modeling, and analysis. It is tempting to produce formal specifications and system models using highly-expressive, rich languages. However, in doing so we often quickly lose desirable features, such as decidability of interesting properties, and tractability of scaling to complex systems. It is better to sacrifice the ability of capturing some classes of behavior, than the framework's support for automated analysis and scalability to real systems. Additionally, one should be able to continuously reap the benefits of extra effort in formalization and modeling. Requiring that a system be first fully formalized provides too steep a cost-benefit curve for most real-world applications. Incrementality and composition should aid not only in the scalability of automated analysis, but also in the efficient use of human effort.

This freedom from "a dogmatic view of formal methods that is detached from mainstream software development" [WJJ96] allowed the successes of the field to flourish. In the late 1990s, NASA published a thorough manual for pragmatic formal methods [Cro+95]: "a practitioner's companion" detailing different degrees of formalization, balancing expressivity with tractability in modeling, and automated analysis powered by finite-state models. Shortly after, case studies were published on experiences using lightweight formal methods in the development of the Space Shuttle, the International Space Station, and other spacecraft [Eas+98].

The early 2000s saw the development of new tooling particularly focused on lightweight techniques, such as the Alloy language [Jac02], which focuses on being a *simpler* specification language to enable also automated analysis. Alloy has over 100 case studies of its application in diverse industry needs [All22]. The 2010s and 2020s saw major companies such as Facebook, Google, and Amazon adopt lightweight formal methods in the development of their products [Cal+15; Bor+21; OHe18; Rei+20], and the development of mainstream tooling for verification, such as Facebook's Infer [Cal+15] static analyzer and Amazon's Kani [Van+22] model checker. These tools have been instrumental in bringing lightweight formal methods to places that before did not directly employ formal verification, such as widely used open source projects, and products of smaller companies [Dis+19; Kum23].

The success of these tools can largely be attributed to hiding the complexity of model-based formal methods. Both Infer and Kani allow a user to annotate existing code with properties of the program's state that should hold before, or after the execution of subroutines. The code and annotations are then used to automatically produce state models, which are analyzed using either computational logics, or symbolic model checking. There are, however, limitations to this approach. Firstly, they can produce models only for certain languages, such as Java/C++, or Rust. Secondly, they only model details of the program, when modern software behavior can rely on external features such as networking, and interactions with hardware. Finally, they naturally require access to source code, when modern software might need to rely on closed source, or proprietary dependencies.

1.3 Automata Learning

Beyond their use as inputs to model-based formal methods, having formal models of a system's behavior allows us to gain a deeper understanding of what is otherwise an effectively *opaque* system. Even if a domain expert such as the system's developer claims to have an understanding of the system, producing a formal model requires a more complete picture, describing precisely what happens on every input, on every state of the system. This includes naturally, modeling unintended behavior such as latent flaws in the system. It is then unreasonable to expect that the expert is aware of all of the behavior of the system, including its possible flaws or bugs. This problem leads us to a natural question: Where do the formal models come from?

Model learning is an area of lightweight formal methods that focuses on automatically *learning* formal behavioral models with correctness guarantees, usually by observing or actively interacting with the system being modeled. A large portion of model-based formal methods use *automata* as their models. Automata are long-standing structures in theoretical computer science, classically used to formalize the limits of computation. Essentially, an automaton is a state machine that computes a *formal language*: a potentially infinite set of strings. Automata are particularly good for expressing how different actions can change the state of a system, and prove properties about such behavior. Different automata types support different features to capture properties of interest, such as timed behavior, concurrency, memory, and input-output mechanics. As such, model learning is vastly dominated by applying the theoretical work of *automata learning* to learn models of systems [Vaa17].

Automata learning has its roots in the late 1960s with the work of Gold [Gol67], who modeled the learning problem as an interaction between a learner, and a teacher. The teacher hides a specific automaton, and allows the learner to ask *membership queries*: whether a given string is or is not computed by said automaton. Gold showed that even automata for simple types of languages cannot be fully learned with a finite number of such questions. However, *in the limit*, exact learnability is possible for so-called regular languages.

Doing so, however, is naturally impractical: "In the limit, we are all dead", as

put by Dana Angluin. Instead, in the late 1980s Angluin [Ang87] contributed the $L^\star$ learning algorithm, powered by a *Minimally Adequate Teacher* (MAT), which is strengthened with the ability to answer also *equivalence queries*. These allow the learner to ask if a hypothesis model is correct, and if not, receive a counterexample string. A learner executing the $L^\star$ algorithm is then able to learn a correct automaton in finite time. Furthermore, it is guaranteed to learn the *minimal* automaton equivalent to the hidden target, in a polynomial number of queries.

Although Angluin presents the $L^\star$ algorithm from a theoretical standpoint, it is quickly put to use in supporting model-based formal methods. In the 1990s, Peled et al. [PVY99] presented the technique of *closed-box model checking*, where model checking and automata learning are used in a symbiotic cycle to verify a closed-box automaton. The following decades saw the development of more efficient algorithms, such as KV [KV94, Section 8.3], TTT [IHS14], and $L^\sharp$ [Vaa+22], as well as algorithms for learning richer types of automata, such as timed automata [HJM20], input-output automata [SG09; vHee14], register automata [Cas+16], weighted automata [Van+20], and many more [Moe+17; DD17; VSS20; Van+21].

These algorithms allowed us to "open the closed box", bringing new avenues in verification. They have been used to directly learn models of a plethora of opaque systems, such as passports [ASV10], bank cards [ARP13], networks and their protocols [FJV16; Fit+17; Fit+20; Moe+25], processor caches [Vil+20], and computer memory vulnerabilities [Nas+22]. Automata learning has also supported lightweight formal methods in other ways, such as improving compositional verification with assume-guarantee reasoning [Păs+08], enabling the verification of neural networks [Khm+21], and more [ASZ21].

Recent work on automata learning has focused on taking its applicability and robustness further: the field of *adaptive* automata learning aims to make learning algorithms that succeed even in suboptimal learning environments [GPY02; Cha+08; HMV18]. Algorithms have been developed for learning with partial teachers, where we may not always be able to observe certain behaviors [LN12; Moe+23], or teachers that incur a level of noise and inconsistency in their responses [Khm+22]. Incremen-

tal algorithms are now capable of *reusing* past knowledge of a system, efficiently detecting the areas in which it might have changed, and reliably updating their models without having to relearn existing behaviors [DMD19; KJR25].

Building on this trajectory, this thesis advances the application of automata learning to lightweight formal methods, addressing challenges across language, modeling, and analysis through the following contributions:

– The PROGNOSIS learning tool for closed-box analysis of network protocols. By combining automata learning with program synthesis, PROGNOSIS has been able to learn models of closed-box network protocol implementations, e.g. QUIC and TCP, in varying granularity, from plain state machines to richer models that capture data flow across packet exchanges. These models were used to uncover shortcomings of the specification of the protocol itself, and vulnerabilities in globally used implementations.

– The IKV adaptive learning algorithm, and the C$\mathfrak{E}$AL adaptive automata learning framework. Together, they address a fundamental practical barrier to deploying automata learning: real-world systems evolve over time and exhibit noisy behavior, yet classical algorithms assume a fixed, deterministic target and must restart from scratch when the system changes. IKV and C$\mathfrak{E}$AL enable efficient, incremental learning that adapts to changes in the target system without discarding prior progress, and that tolerates noisy observations during the learning phase.

– The SWAL algorithm for learning weighted automata over quantitative domains commonly seen in practice, such as costs and bottlenecks. It contributes both theoretical automata characterizations that power learning, and techniques for learning quantitative systems with varying correctness guarantees. Notably, SWAL guarantees state-minimality of the learned automata across a broad range of semirings — a guarantee that prior $L^\star$-style approaches achieve only in restricted settings.

1.4 Thesis Overview

Having situated the contributions of this thesis, in the next subsections we provide an overview of the core technical contributions (Chapters 3 to 6). In Chapter 2 we introduce preliminary concepts in formal language theory, automata theory, and classical algorithms in automata learning. Chapter 7 concludes this thesis with a discussion of future work, including directions that build on collaborative efforts.

1.4.1 Learning Network Protocol Implementations (Chapter 3)

As a primer for the potential of automata learning in lightweight formal methods, Chapter 3 introduces PROGNOSIS, a modular tool offering automated closed-box learning and analysis of models of network protocol implementations (cf. Figure 3.1).

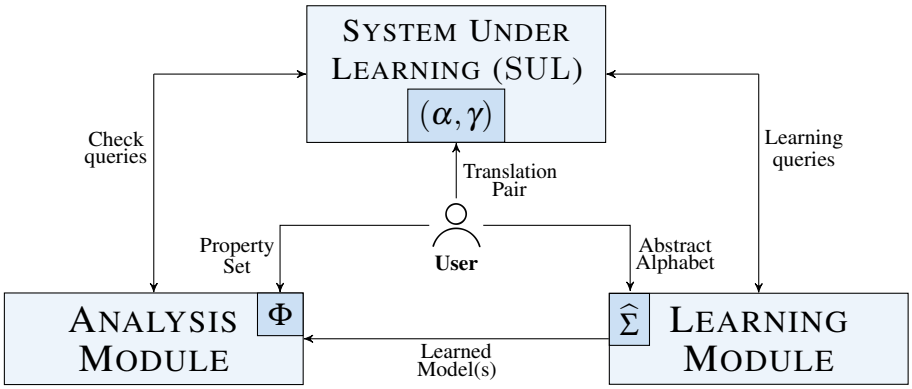


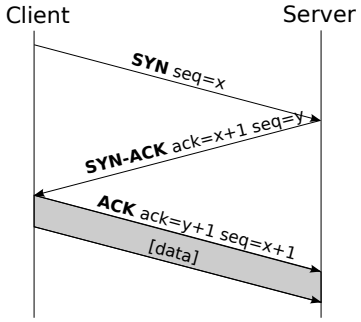
Figure 3.1: PROGNOSIS' modular architecture.

(reproduced from p. 53)

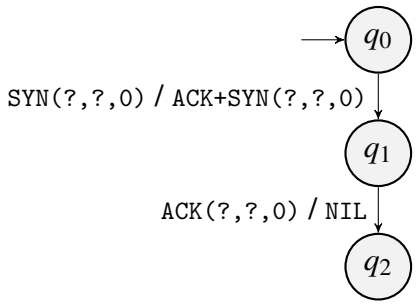
By interacting with the implementation through network packets, PROGNOSIS can learn models that vary in abstraction level: from simple deterministic automata powered by classic automata learning algorithms, to models containing data operations, such as register updates through program synthesis (cf. Figure 3.3).

PROGNOSIS uses these models to unlock a variety of analysis techniques: model checking temporal properties, computing differences between models of two implementations, or improving testing via model-based test generation. PROGNOSIS is modular and easily adaptable to different protocols (e.g., TCP and QUIC) and their implementations. Configurable abstraction levels help with scalability: coarser models can be learned quickly and, when a potential bug is found at the model level,

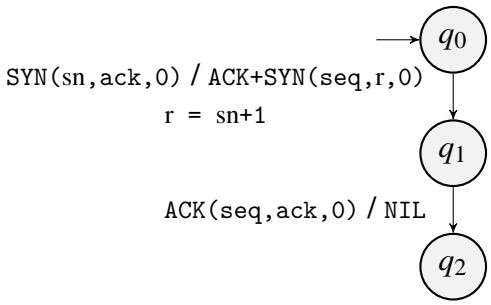
PROGNOSIS creates concrete traces to check whether the bug manifests in the actual implementation or is a false positive that can be used to refine the model.



(a) The TCP 3-way handshake [LU13].



(b) Learned model (fragment).



(c) Synthesized model (fragment) with register r.

Figure 3.3: TCP handshake: protocol, learned model, and synthesized model.

(reproduced from p. 57)

Being entirely closed-box, PROGNOSIS requires neither access to source code nor a manually written formal specification. We demonstrate this by learning models of (parts of) three QUIC implementations — Quiche (Cloudflare), Google QUIC, and Facebook mvfst — and analyzing the differences between them. Our analysis provided insights into different design choices and uncovered critical bugs acknowledged by the developers (cf. Table 3.1).

Table 3.1: Summary of PROGNOSIS results on QUIC implementations.

Target	Model	Bug Type	Detection
QUIC RFC	Learned	Underspecification	Equiv. Check
Cloudflare Quiche	Learned + Synth.	—	—
Google QUIC	Learned + Synth.	Unimpl. Behavior	Equiv. Check + Manual
Facebook mvfst	Partial	Security Vulnerability	Nondet. Check
Quic-Tracker	Learned	Logic Error	Manual Analysis

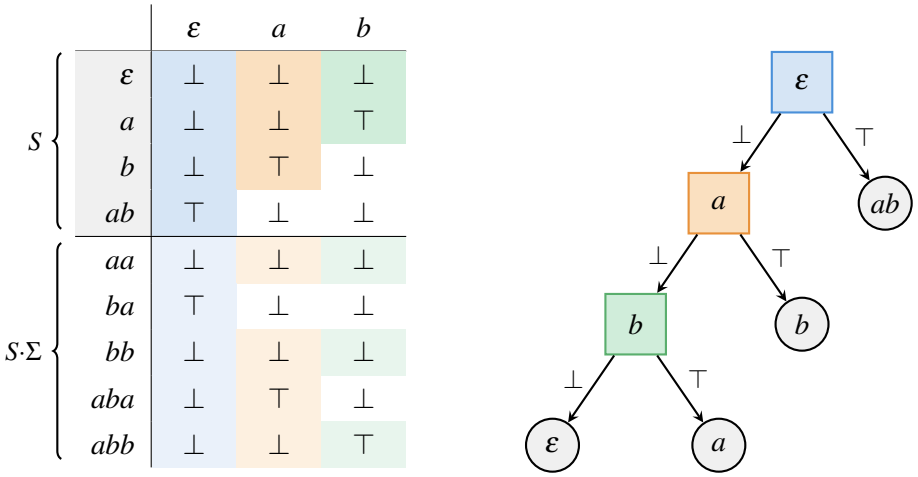
(reproduced from p. 77)

The contributions of this chapter have been published in the following paper:

Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. "Prognosis: Closed-Box Analysis of Network Protocol Implementations". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21: ACM SIGCOMM 2021 Conference. Virtual Event USA: ACM, Aug. 9, 2021, pp. 762–774. ISBN: 978-1-4503-8383-7. DOI: 10.1145/3452296.3472938

1.4.2 Learning Evolving Systems (Chapter 4)

In Chapter 4, motivated by the application of PROGNOSIS to quickly evolving software, we contribute the IKV adaptive learning algorithm. IKV extends the KV learning algorithm [KV94] to be able to handle systems that change over time. While state-of-the-art adaptive learning algorithms were based on L* and its observation tables [DMD19], we present the first adaptive algorithm based on KV's classification trees, greatly reducing redundant information and queries.

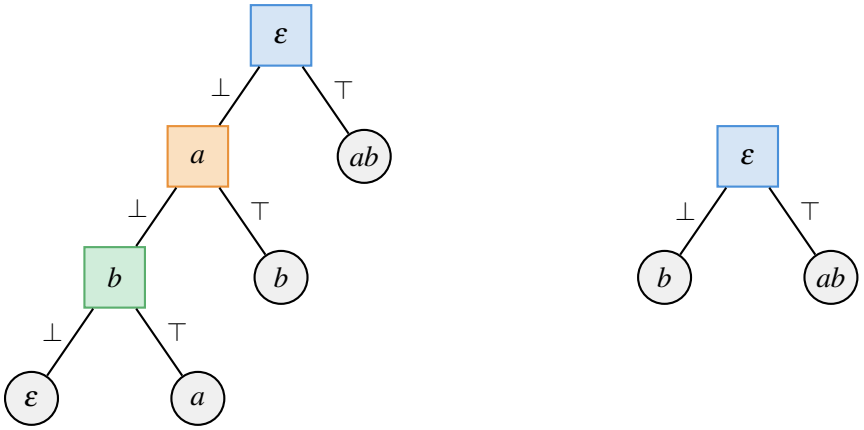


(a) L* observation table (b) KV classification tree

Figure 1.1: Language information cover of observation table vs classification tree.
(reproduced from Example 4.1.2, p. 86)

The key observation in the IKV algorithm is that to be able to efficiently learn *differences* when the system changes, we must have a way of *pruning* the tree from outdated behavior, rather than strictly growing it. IKV is able to do this through a

new efficient subroutine that minimally rechecks if the tree's structure is still valid, and if not, what cuts need to be done to make it consistent with the new target.



(a) Classification tree for $\mathcal{L}$ ($|w|_a$ *is odd* $\wedge$ $|w|_b$ *is odd*)

(b) Pruned classification tree for $\mathcal{L}'$ ($|w|_a$ *is odd*)

Figure 1.2: Pruning a previous tree (a) learned for language $\mathcal{L}$ into a tree (b) containing only information consistent with the new (closed-box) target language $\mathcal{L}'$.

(reproduced from Example 4.2.1, p. 96)

We implement IKV in the LearnLib library [FHS25], and evaluate it on large examples, to which we make small adjustments between two runs of the algorithm. In these experiments our algorithm significantly outperforms both the classic KV learning algorithm and previous state-of-the-art adaptive algorithms.
This chapter's contributions have been published in the paper:

Tiago Ferreira, Gerco Van Heerdt, and Alexandra Silva. "Tree-Based Adaptive Model Learning". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra Van Den Bos. Vol. 13560. Cham: Springer Nature Switzerland, 2022, pp. 164–179. ISBN: 978-3-031-15628-1 978-3-031-15629-8. DOI: 10.1007/978-3-031-15629-8_10

1.4.3 Learning Conflicting Systems (Chapter 5)

In Chapter 5 we take adaptive automata learning further with a new alternative to the MAT framework. While IKV represents a significant improvement to the efficiency of adaptive automata learning, it targets only the specific case of learning evolving systems. The crux of the issue is that classical automata learning algorithms cannot

easily handle *conflict* in the observation data (different outputs observed for the same inputs). This inherent inability to recover after a conflict impairs not only learning target systems that may evolve, but also learning when noise is present, or observations are unreliable.

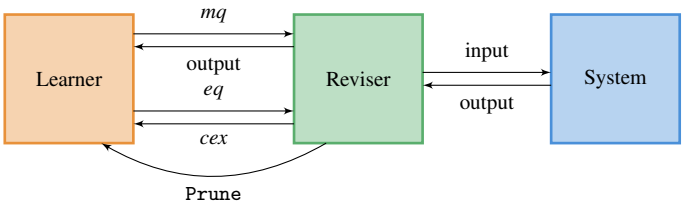
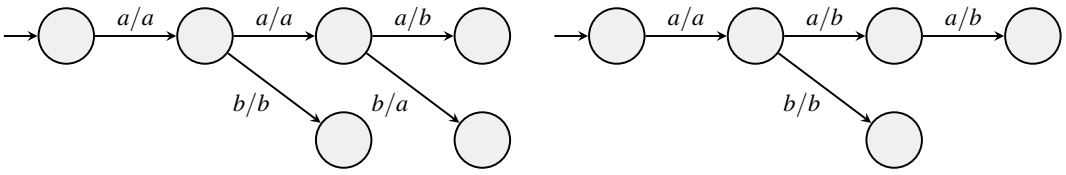


Figure 5.1: Simplified view of the CℰAL framework. See Figures 5.3 and 5.4 for details.
(reproduced from p. 117)

We propose instead the Conflict-Aware Active Automata Learning (CℰAL) framework, to enable handling conflicting information during the learning process. The core idea is to consider the so-called observation tree (see Figure 1.3) as a first-class citizen in the learning process. Though this idea is explored in recent work [Vaa+22], we take it to its full effect by using it to minimize the number of queries needed to learn in the face of conflicts.



$\mathcal{L} = \{(aaa, aab), (aab, aaa), (ab, ab)\}$ $\mathcal{L} = \{(aaa, abb), (ab, ab)\}$

Figure 1.3: Two observation trees and their languages.
(reproduced from Example 5.2.4, p. 120)

The observation tree powers a new agent, the Reviser, which is responsible for ensuring the learner has only a consistent view of the system being learned at any given point. It is responsible for realizing the learner's need for membership and equivalence queries, and ensure the learner's data structure remains conflict-free. In this way, any existing MAT learning algorithm can be used in CℰAL, while enjoying a more robust learning environment in the face of noise or system evolutions.

(a) Learner's view **(b)** System's view

Figure 5.2: The Reviser's interfaces.

(reproduced from p. 118)

We evaluate CƐAL in a large set of benchmarks, covering 36 realistic targets and over 18,000 scenarios, totalling close to 2 million learning runs. The results show that CƐAL achieves an overall success rate of 95.5% against MAT's 79.5%, provides the most efficient correct configuration in 70% of scenarios, and successfully learns targets that no MAT configuration could handle.

The contributions of this chapter have been published in the paper:

Tiago Ferreira, Léo Henry, Raquel Fernandes Da Silva, and Alexandra Silva. "Conflict-Aware Active Automata Learning". In: *Electronic Proceedings in Theoretical Computer Science* 390 (Sept. 30, 2023), pp. 150–167. ISSN: 2075-2180. DOI: 10.4204/EPTCS.390.10

1.4.4 Learning Quantitative Systems (Chapter 6)

In Chapter 6 we present SWAL, an SMT-based algorithm for learning nondeterministic *weighted* automata (WFAs) as a practical and robust alternative to Hankel/L*-style methods. Although L*'s theory is derived from a characterization specific to DFAs and regular languages, the majority of weighted automata algorithms are a result of attempts to extend it to the quantitative, non-deterministic domain [Bol+09; Van+20]. This disconnect often leads to issues such as non-minimality, and non-termination.

$$\iota \triangleq \begin{pmatrix} 4 & 5 \end{pmatrix} \qquad \delta^a \triangleq \begin{pmatrix} 6 & \infty \\ \infty & 4 \end{pmatrix}$$

$$\delta^b \triangleq \begin{pmatrix} 6 & 8 \\ \infty & 12 \end{pmatrix} \qquad \lambda \triangleq \begin{pmatrix} \infty \\ 3 \end{pmatrix}$$

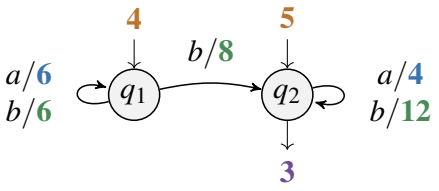


Figure 1.4: A 2-state WFA over the Tropical semiring $\mathbb{S} \triangleq \langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0 \rangle$.

(reproduced from Example 6.1.3, p. 144)

Rather than maintaining and closing an observation table in the style of L*, SWAL casts the learning problem directly as a constraint-solving problem. Inspired by a coalgebraic view of WFAs, we derive a system of equations over the given semiring whose solutions correspond exactly to WFAs consistent with the words queried so far. Learning then proceeds by incrementally strengthening this equation system in response to counterexamples and invoking an SMT solver to find a satisfying assignment. By searching for the smallest consistent automaton at each step, the algorithm produces minimal WFAs by construction, a guarantee that L*-based methods can only offer for restricted classes of semirings. We show that the resulting equation systems fall into decidable SMT theories for several relevant semirings, including all finite semirings and infinite ones such as the tropical semiring.

$$\Phi(n, O) \triangleq \underbrace{\bigwedge_{i=1}^{n} \bigwedge_{aw \in \mathsf{suffixes}(O)} f_i^{aw} = (\delta_{i,1}^a \odot f_1^w) \oplus \ldots \oplus (\delta_{i,n}^a \odot f_n^w)}_{\delta \text{ stabilizes } \mathcal{F} \text{ for } O \text{ (Definition 6.3.6.2)}}$$
$$\wedge \underbrace{\bigwedge_{w \in O} mq(w) = (\iota_1 \odot f_1^w) \oplus \ldots \oplus (\iota_n \odot f_n^w)}_{\iota \text{ spans } \mathcal{L} \text{ for } O \text{ (Definition 6.3.6.3)}}$$

Figure 6.1: Equation system $\Phi(O, n)$ characterizing the existence of n-state O-correct WFAs. Here $(f_i^w)_{i \in n, w \in \mathsf{suffixes}(O)}$, $(\delta_{i,j}^a)_{a \in \Sigma, (i,j) \in n \times n}$, and $(\iota_i)_{i \in n}$ are $\mathbb{S}$-valued variables.
(reproduced from p. 154)

Our experimental evaluation spans over 5000 benchmarks across the Boolean, tropical (bounded and unbounded), and bottleneck semirings, and studies the impact of incremental SMT solving and of alternative semiring encodings on the algorithm's overall performance. The results show that SWAL is capable of learning numerous minimal WFAs over both finite and infinite semirings, vastly outperforms a naive baseline, and is competitive with the state of the art while producing (sometimes exponentially) smaller models and requiring fewer teacher interactions.
The contributions of this chapter have been published as the following paper:

Tiago Ferreira, Kevin Batz, and Alexandra Silva. "SMT-Based Active Learning of Weighted Automata". In: *Computer Aided Verification*. Ed. by Eva Darulova, An-

thony W. Lin, and Philipp Rümmer. Vol. 16683. Cham: Springer Nature Switzerland, 2026, pp. 307–330. DOI: 10.1007/978-3-032-32526-6_15

1.4.5 Other Contributions

Beyond the chapters of this thesis, the research programme pursued during this PhD has led to several collaborative works that extend and complement the contributions presented here, as listed below. On the application side, PROGNOSIS was deployed in collaboration with Galois, Inc. to analyze real-world protocol targets as part of DARPA-funded network assurance work. On the theoretical side, we contributed to broadening the scope of automata learning for networking and verification: developing new expressive automata models and learning algorithms for full network policies, extending automata-based network programming languages with quantitative weighted semantics over arbitrary semirings, and leveraging large language models to extract state machines from informal protocol specifications.

Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. "Symbolic Register Automata". In: *Computer Aided Verification*. Ed. by I. Dillig and S. Tasiran. Vol. 11561. Cham: Springer International Publishing, 2019, pp. 3–21. ISBN: 978-3-030-25539-8 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_1

Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. "Active Learning of Symbolic NetKAT Automata". In: *Proceedings of the ACM on Programming Languages* 9 (PLDI June 10, 2025), pp. 1119–1142. ISSN: 2475-1421. DOI: 10.1145/3729295

Noga H. Rotman, Tiago Ferreira, Hila Peleg, Mark Silberstein, and Alexandra Silva. *RFSeek and Ye Shall Find*. Under review at ANRW. Sept. 12, 2025. DOI: 10.48550/arXiv.2509.10216. arXiv: 2509.10216 [cs.NI]. Pre-published

Emmanuel Suárez Acevedo, Tiago Ferreira, Kevin Batz, Oliver Bøving, Nate Foster, and Alexandra Silva. "Weighted NetKAT: A Programming Language for Quantitative Network Verification". In: *Proceedings of the ACM on Programming Languages* 10 (PLDI June 8, 2026), pp. 1788–1811. ISSN: 2475-1421. DOI: 10.1145/3808318

Artifact Availability. The source code, implementations, and replication packages associated with this thesis are available in the accompanying archival artifact [Fer26]. Unless otherwise indicated, the artifact and original contributions are released under the MIT License. This does not supersede the existing licences of bundled dependencies or incorporated third-party source code.

Chapter 2

Preliminaries

In this chapter we recall preliminary knowledge central to this thesis. In particular, we establish commonly used notation, and advance to introducing formal languages of different shapes, the automata that support their computation, and their seminal learning algorithms.

As foundation, we denote the empty set by $\varnothing \triangleq \{\}$, the unit set by $\mathbb{1} \triangleq \{\bullet\}$, and the set of Booleans by $\mathbb{2} \triangleq \{\bot, \top\}$. We write functions from X to Y as Y^X, or more commonly, $X \to Y$. We write the *copairing*, or disjoint union of sets X and Y as $X + Y$. We will often use the set $X + \mathbb{1}$ to represent *optional* or *nullable* values in X.

2.1 Formal Languages

Before we consider automata and their learning, we must acquaint ourselves with *formal languages*, the underlying objects that automata *compute*. From the bottom up, an *alphabet* is a set of atomic symbols, such as $\Sigma = \{a, b, c\}$, over which we can form words: sequences of symbols such as $w = aabcab$. Classically, and for the structures studied in this thesis, alphabets and words are strictly finite. We denote the empty word by ε, the i-th (0 indexed) symbol of a word w by $w[i]$, and the *range* of symbols from i to j in word w by $w[i..j]$. We denote the concatenation of words w and w' by $w \cdot w'$, or simply by their juxtaposition ww'. A *language* is then a (potentially infinite) set of words, such as $\mathcal{L} = \{\varepsilon, a, ba, bcacab\}$. We can then perform operations on formal languages:

Definition 2.1.1 (Language Operations)**.** *Given languages $\mathcal{L}, \mathcal{L}'$ over an alphabet Σ,*

we define their union, concatenation, and Kleene star as follows:

$$\mathcal{L}_1 \cup \mathcal{L}_2 \triangleq \{w \mid w \in \mathcal{L}_1 \vee w \in \mathcal{L}_2\}$$
$$\mathcal{L}_1 \cdot \mathcal{L}_2 \triangleq \{w \cdot v \mid w \in \mathcal{L}_1 \wedge v \in \mathcal{L}_2\}$$
$$\mathcal{L}^\star \triangleq \bigcup_{i \in \mathbb{N}} \mathcal{L}^i \quad \textit{where} \quad \mathcal{L}^0 \triangleq \{\varepsilon\} \ \textit{and} \ \mathcal{L}^{i+1} \triangleq \mathcal{L}^i \cdot \mathcal{L}$$

△

The star operation is often applied to alphabets to mean the language of all words of a given alphabet. For example, $\Sigma^\star = \{\varepsilon, a, b, c, aa, ab, ba, \dots\}$. Similarly, $\Sigma^+ \triangleq \Sigma^\star \setminus \{\varepsilon\}$ denotes the set of all nonempty words. It is common to say then that a language is just a specific set of these possible words ($\mathcal{L} \in \mathcal{P}(\Sigma^\star)$), or equivalently a *function* $\Sigma^\star \to 2$ determining whether any given string is in said language or not.

Despite their simplicity, formal languages can capture a large range of behaviors. In fact, we know that there is a hierarchy of languages, from those enjoying a plethora of efficient, decidable properties, to those aligned with the power of general computation, to those beyond what can be computed at all [CS63]. On the simpler side of this hierarchy are the *regular* languages.

Definition 2.1.2 (Regular Language)**.** *The* regular languages *over* Σ *are the smallest class* $\mathcal{P}(\Sigma^\star)^{reg} \subseteq \mathcal{P}(\Sigma^\star)$ *containing* $\varnothing$, $\{\varepsilon\}$, *and* $\{a\}$ *for each* $a \in \Sigma$, *and closed under union* ($\mathcal{L}_1 \cup \mathcal{L}_2$)*, concatenation* ($\mathcal{L}_1 \cdot \mathcal{L}_2$)*, and Kleene star* ($\mathcal{L}^\star$)*.* △

Regular languages (and their generalizations) will be the main semantic object of this thesis: their simplicity allows for algorithms underpinning automated analysis in lightweight formal methods, and they can still capture behaviors of great relevance, as we will see in Chapter 3.

2.2 Automata Theory

Intuitively, an automaton is a state machine that *computes* a language. Of classical importance are *Deterministic Finite Automata* (DFAs), which given an input word, compute whether it is part of the given language or not. A DFA does so by, starting

from a *start state*, consuming the symbols in the word one-by-one, changing its state with each symbol. If after processing a word the automaton ends in an accepting state, the word is accepted, and otherwise it is rejected. It is *deterministic* because each word elicits exactly one path of states visited, and it is *finite* because the automaton must have only a finite set of such states.

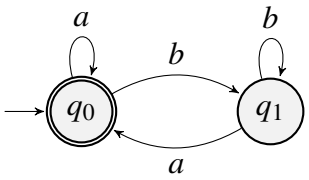
Definition 2.2.1 (Deterministic Finite Automaton)**.** *Formally, a* deterministic finite automaton *(DFA) is a tuple* $\mathcal{A} \triangleq \langle Q, q_0, \Sigma, \delta, F \rangle$ *such that:* Q *is a finite set of states;* $q_o \in Q$ *is the* start *or initial state;* Σ *is the alphabet of the automaton;* $\delta \colon Q \times \Sigma \to Q$ *is the* transition *function;* $F \subseteq Q$ *is the set of* accepting *or final states.*
The semantics of a DFA are given by its language $[\![\mathcal{A}]\!](q_0) \colon \Sigma^\star \to \mathbb{2}$ *defined as:*

$$[\![\mathcal{A}]\!](q)(\varepsilon) \triangleq q \in F \qquad [\![\mathcal{A}]\!](q)(a \cdot w) \triangleq [\![\mathcal{A}]\!](\delta(q,a))(w)$$

Where clear we omit the state parameter from $[\![\mathcal{A}]\!](q_0)$ *writing simply* $[\![\mathcal{A}]\!]$. △

Often, we depict automata as annotated graphs, where states (Q) are represented as vertices, and labeled edges represent how different symbols of the alphabet (Σ) change the state of the automaton (δ). The start state (q_0) is depicted by a leading edge, and final states (F) by double lined vertices.

Example 2.2.2. *Consider the language* $\mathcal{L} \triangleq \{\varepsilon, w \cdot a \mid w \in \Sigma^\star\}$ *over alphabet* $\Sigma \triangleq \{a, b\}$*. This is the language of words that are empty or end in a. A two state DFA* $\mathcal{A}$ *for* $\mathcal{L}$ *is depicted below (left), as well as example word computations (right).*



$$\varepsilon : q_0 \in F \qquad \rightsquigarrow \top$$
$$ab : q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \notin F \qquad \rightsquigarrow \bot$$
$$aba : q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_0 \in F \qquad \rightsquigarrow \top$$

It is sometimes convenient to have labels for the two possible single-state DFAs over an alphabet Σ. We denote $\mathcal{A}_\Sigma^\top$ the single (accepting) state automaton with language $[\![\mathcal{A}_\Sigma^\top]\!] = \Sigma^\star$, and $\mathcal{A}_\Sigma^\bot$ the single-state automaton with language $[\![\mathcal{A}_\Sigma^\bot]\!] = \varnothing$.

We might naturally wonder, given that there are languages that are not computable at all, what languages can a DFA compute? It turns out they are precisely the regular languages.

Definition 2.2.3 (Recognizability)**.** *A formal language $\mathcal{L}$ is said to be* recognizable *if there exists a DFA $\mathcal{A}$ such that $\mathcal{L} = [\![\mathcal{A}]\!]$. The set of recognizable languages is denoted $\mathcal{P}(\Sigma^\star)^{rec} \subseteq \mathcal{P}(\Sigma^\star)$.* △

Theorem 2.2.4 (Kleene [Kle56])**.** *Let $\mathcal{L} \subseteq \Sigma^\star$ be a formal language. Then $\mathcal{L}$ is regular if and only if it is recognizable. Formally, $\mathcal{P}(\Sigma^\star)^{reg} = \mathcal{P}(\Sigma^\star)^{rec}$.*

This connection leads us to an interesting question: every DFA then recognizes a regular language, and every regular language is recognized by a DFA. However, a DFA has precisely one language, but a regular language can have multiple DFAs of different sizes. What is then *the* canonical DFA for a given regular language? A first good principle is that said DFA should be as small as possible, i.e., minimal.

Definition 2.2.5 (Minimality)**.** *A DFA is* minimal *if and only if there does not exist a DFA with fewer states for the same language.* △

However, we may still wonder: what if I have two different minimal DFAs for the same language? It turns out this cannot be. This is due to an important result of Myhill and Nerode [Ten+57] establishing a necessary and sufficient characterization of regular languages.

Definition 2.2.6 (Myhill–Nerode Relation)**.** *Given a formal language $\mathcal{L}$, and a pair of words $x, y \in \Sigma^\star$ we say $x \cong_{\mathcal{L}} y$ if and only if there is no distinguishing extension $z \in \Sigma^\star$ for them. Formally:*

$$x \cong_{\mathcal{L}} y \iff \forall z \in \Sigma^\star.\ \mathcal{L}(x \cdot z) = \mathcal{L}(y \cdot z)$$

△

Theorem 2.2.7 (Myhill–Nerode [HU69, Theorem 3.1])**.** *A language $\mathcal{L}$ is regular, if and only if the equivalence relation $\cong_{\mathcal{L}}$ has finite index ($|\cong_{\mathcal{L}}| \in \mathbb{N}$).*

For some intuition on this connection, note that the Myhill–Nerode relation captures precisely what it means for two strings to access the same state in a minimal automaton, just written at the level of the language. If two strings are distinguishable, i.e., there exists an extension $z \in \Sigma^\star$ such that $\mathcal{L}(xz) \neq \mathcal{L}(yz)$, then it must be that the strings x, y end up in different states of the automaton. Otherwise, as the automaton is deterministic, if x, y lead to the same state, their continuations would follow the same path from that same state, into also a same state, and both xz, yz would then be in the language, or both not in the language. Similarly, if two strings x, y are access strings of different states in a minimal automaton, it must then be that there exists some continuation z for which $\mathcal{L}(xz) \neq \mathcal{L}(yz)$, otherwise, the states of strings x, y would produce exactly the same extensions, and therefore the automaton would not be minimal: one of the states is redundant.

The statement of the Myhill–Nerode theorem is then perhaps not surprising at all: a language is regular iff the Myhill–Nerode relation has *finite* index, i.e., iff it has a minimal deterministic *finite* automaton. In a way, Myhill–Nerode gives us a language-level characterization of what it means for a language to have a DFA (and therefore to be regular by Theorem 2.2.4).

Furthermore, we can exploit this connection between language and structure to produce *the* canonical automaton for the language. We denote by $[w]_{\cong_\mathcal{L}}$ the equivalence class of w in $\cong_\mathcal{L}$.

Definition 2.2.8 (Canonical DFA)**.** *For a given regular language $\mathcal{L} \in \mathcal{P}(\Sigma^\star)^{reg}$, the* canonical *DFA $\mathcal{A}_\mathcal{L} \triangleq \langle Q, q_0, \Sigma, \delta, F \rangle$ is defined by:*

$$Q \triangleq \{[w]_{\cong_\mathcal{L}} \mid w \in \Sigma^\star\} \qquad q_0 \triangleq [\varepsilon]_{\cong_\mathcal{L}}$$
$$F \triangleq \{[w]_{\cong_\mathcal{L}} \mid \varepsilon \in [w]_{\cong_\mathcal{L}}, w \in \Sigma^\star\} \qquad \delta([w]_{\cong_\mathcal{L}})(a) \triangleq [w \cdot a]_{\cong_\mathcal{L}}$$

△

Theorem 2.2.9 ([HU69, Theorem 3.2])**.** *For a given regular language $\mathcal{L}$, the* canonical *DFA $\mathcal{A}_\mathcal{L}$ is* minimal *and* unique *(up to state renaming).*

As we will see in Section 2.3, the guarantees afforded by the Myhill–Nerode

theorem, from finiteness of Myhill–Nerode classes to their contribution to producing minimal DFAs, form the central theory of the majority of active automata learning algorithms. We have covered these matters in a level of detail tailored to this role, and point the reader to Hopcroft and Ullman [HU69, Section 3.2] for a more extensive presentation of the Myhill–Nerode Theorem and its consequences.

2.2.1 Transducers

The types of automata discussed above belong to a group of automata called *acceptors*, as given an input word they either *accept* it or not. There are however, other types of automata: *transducers* are automata that given an input word, produce instead an *output* word. They can be thought of as language translators. These types of automata are crucial for applications of automata learning, as they capture the input-output behavior of reactive systems: The user performs an action (input), the machine responds by producing something, displaying an error, etc. (output), and moving to a next state. This is precisely how Mealy machines [Mea55] behave.

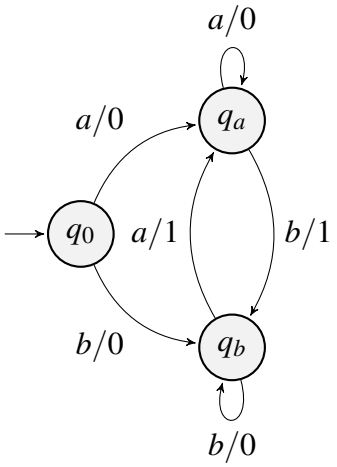
Definition 2.2.10. *A* Mealy machine *is a tuple* $\mathcal{M} \triangleq \langle Q, q_0, \Sigma, \Gamma, \delta, \lambda \rangle$ *where:* Q *is a finite set of states;* q_0 *is the* initial *state;* Σ *is a finite* input *alphabet;* Γ *is a finite* output *alphabet;* $\delta \colon Q \times \Sigma \to Q$ *is a transition function; and* $\lambda \colon Q \times \Sigma \to \Gamma$ *is the* output *function. The semantics of* $\mathcal{M}$ *is* $[\![\mathcal{M}]\!](q_0) \colon \Sigma^\star \to \Gamma^\star$ *defined by:*

$$[\![\mathcal{M}]\!](q)(\varepsilon) \triangleq \varepsilon \qquad [\![\mathcal{M}]\!](q)(a \cdot w) \triangleq \lambda(q,a) \cdot [\![\mathcal{M}]\!](\delta(q,a))(w)$$

Where clear, we omit the initial state parameter $[\![\mathcal{M}]\!](q_0)$*, writing only* $[\![\mathcal{M}]\!]$. △

The definition above makes clear the parallels of Mealy machines and DFAs. A Mealy machine can be considered to be a DFA endowed with an extra alphabet, from which it draws outputs on each transition, instead of on a final visited state.

Example 2.2.11. *Consider the* edge detector *Mealy machine over input alphabet* $\Sigma \triangleq \{a,b\}$ *and output alphabet* $\Gamma \triangleq \{0,1\}$. *This Mealy machine outputs* 1 *when the input symbol is different to the previous one, and* 0 *otherwise. Below we give the Mealy machine pictorially (left) and example computations (right).*



$$ab : q_0 \xrightarrow{a/0} q_a \xrightarrow{b/1} q_b \rightsquigarrow 01$$

$$aa : q_0 \xrightarrow{a/0} q_a \xrightarrow{a/0} q_a \rightsquigarrow 00$$

$$aba : q_0 \xrightarrow{a/0} q_a \xrightarrow{b/1} q_b \xrightarrow{a/1} q_a \rightsquigarrow 011$$

$$abb : q_0 \xrightarrow{a/0} q_a \xrightarrow{b/1} q_b \xrightarrow{b/0} q_b \rightsquigarrow 010$$

$$bbb : q_0 \xrightarrow{b/0} q_b \xrightarrow{b/0} q_b \xrightarrow{b/0} q_b \rightsquigarrow 000$$

Notice that Mealy machines satisfy a *causal* property: Every input symbol produces precisely one output symbol, meaning that an input word is transformed into an output word of the same size, and every prefix of said input produces the same size prefix of the respective output. As such, without loss of generality, some presentations of Mealy machine consider instead a semantics $[\![\mathcal{M}]\!] \colon \Sigma^+ \to \Gamma$, where given a nonempty input string, the Mealy machine returns only the last output symbol produced. Because of causality, one can simply run all the prefixes of the input word to get the full output word, and therefore the definitions are equivalent.

Such a semantics supports a generalization of the Myhill–Nerode relation to Mealy machines in which equivalence is defined directly on input words. This construction goes back to Nerode's characterization of finite-state input–output transformations using equivalence classes of input histories [Ner58], while we follow the finite-word formulation of van Heerdt [vHee14, Section 3.2]. The resulting relation inherits the same guarantees of minimality and uniqueness as in the DFA setting. Concretely, we regard the behavior of a Mealy machine as a "language" $\mathcal{L} \colon \Sigma^+ \to \Gamma$, and adapt the usual definition of the Myhill–Nerode relation accordingly.

Definition 2.2.12. *Given a language $\mathcal{L} \to \Sigma^+ \to \Gamma$, and pair of words $x, y \in \Sigma^\star$, we say $x \cong_{\mathcal{L}} y$ iff there is no distinguishing extension $z \in \Sigma^+$ for them. Formally:*

$$x \cong_{\mathcal{L}} y \iff \forall z \in \Sigma^+.\ \mathcal{L}(x \cdot z) = \mathcal{L}(y \cdot z)$$

△

Producing the canonical Mealy machine is also done equivalently: We just swap the set of final states by transition outputs, which are taken from the language.

Lemma 2.2.13. *Given a regular language* $\mathcal{L}\colon \Sigma^+ \to \Gamma$*, its canonical Mealy machine* $\mathcal{M}_{\mathcal{L}} \triangleq \langle Q, q_0, \Sigma, \Gamma, \delta, \lambda \rangle$ *is* minimal *and* unique *up to state renaming, where:*

$$Q \triangleq \{[w]_{\cong_{\mathcal{L}}} \mid w \in \Sigma^\star\} \qquad q_0 \triangleq [\varepsilon]_{\cong_{\mathcal{L}}}$$
$$\delta([w]_{\cong_{\mathcal{L}}}, a) \triangleq [w \cdot a]_{\cong_{\mathcal{L}}} \qquad \lambda([w]_{\cong_{\mathcal{L}}}, a) \triangleq \mathcal{L}(w \cdot a)$$

2.3 Automata Learning

Automata learning can be understood as a collection of algorithms for producing automata models from finite traces, or observations. Passive automata learning concerns itself with producing models consistent with previously gathered, finite sets of traces. Such models may generalize beyond the observed data, but the learner has no means of validating those generalizations against the target system. The resulting model is therefore best understood as a compact hypothesis derived from the available observations, whose accuracy outside them depends on the assumptions and biases of the learning algorithm. For the purposes of verification, and lightweight formal methods, we want instead learning algorithms that are capable of gathering as much information as they deem necessary to provide correctness guarantees, in order to be able to find and capture unforeseen behaviors, such as bugs in edge cases. Such algorithms belong to the group of *active* (or online) automata learning.

The seminal work on active automata learning is Angluin's $L^\star$ algorithm [Ang87] for learning DFAs. It can be understood as exploiting the guarantees of the Myhill–Nerode theorem (Theorem 2.2.7) to produce *canonical* DFAs of a hidden, closed-box language.

2.3.1 The Minimally Adequate Teacher

The learning environment is classically modeled as an interaction between two entities: the *learner* and the *teacher*. Angluin specifies that the $L^\star$ algorithm needs access to a *minimally adequate teacher* to ensure its correctness properties.

Definition 2.3.1 (Minimally Adequate Teacher [Ang87])**.** *A* Minimally Adequate Teacher *(MAT) is capable of answering two types of queries, relative to a target language* $\mathcal{L}\colon \Sigma^\star \to \mathbb{2}$*:*

– **Membership Queries** *of the type* $mq\colon \Sigma^\star \to \mathbb{2}$ *that indicate whether a given word is part of the target language.*

– **Equivalence Queries** *of the type* $eq\colon \mathsf{DFA} \to \mathbb{1} + \Sigma^\star$ *that indicate whether a given hypothesis automaton is correct, and if not, provide a counterexample witnessing this discrepancy.*

△

The learner, on the other hand, is responsible for executing an algorithm that correctly produces the canonical automaton for the target hidden language, while using the least amount of queries as possible.

2.3.2 The Observation Table

The L* algorithm works by repeatedly refining an internal data structure called an *observation table*. The table is the key in determining both the information that is asked of the teacher, and the automaton that a given set of information represents.

Definition 2.3.2 (Observation Table [Ang87])**.** *An observation table* T *is a (curried) function* $S \cup (S \cdot \Sigma) \to E \to \mathbb{2}$ *where* $S \subset \Sigma^\star$ *is a finite set of* access strings*, and* $E \subset \Sigma^\star$ *is a finite set of distinguishing suffixes. The table is filled such that* $T(w)(w') = \mathcal{L}(w \cdot w')$*, via membership queries. The partial application* $T(s)\colon E \to \mathbb{2}$ *gives access to the row of* $s \in S \cup (S \cdot \Sigma)$*.* △

It is often said that the observation table is used to keep track of information on the learned automata, however it is more precise to say that the observation table keeps track of the information learned about the Myhill–Nerode *equivalence classes* (Definition 2.2.8) of the hidden language. In particular, the access strings $x, y \in S$ represent specific equivalence classes, while distinguishing suffixes $z \in E$ provide witnesses to different access strings belonging to different equivalence classes. If $T(x)(z) \neq T(y)(z)$ then by Myhill–Nerode (Theorem 2.2.7) we know that the strings

belong to different equivalence classes, and therefore access different states in the hidden canonical automaton. In this way, the *rows* of the observation table provide a partial view of the Myhill–Nerode equivalence classes.

Once the algorithm determines the table holds "enough" information on the hidden language (i.e., that the table is closed and consistent, which we define later), we can produce the canonical automaton of the respective Myhill–Nerode relation information, in an equivalent manner as in Definition 2.2.8.

Definition 2.3.3. *Given a closed and consistent observation table T, we can produce its minimal consistent automaton $\mathcal{A}_T \triangleq \langle Q, q_0, \Sigma, \delta, F \rangle$ defined by:*

$$Q \triangleq \{T(s) \mid s \in S\} \qquad q_0 \triangleq T(\varepsilon)$$
$$\delta(T(w))(a) \triangleq T(w \cdot a) \qquad F \triangleq \{T(s) \mid s \in S \wedge T(s)(\varepsilon) = \top\}$$

△

Notice that the definition of the table's automaton mirrors that of producing the canonical automaton from a given Myhill–Nerode equivalence relation. The row of a given access string represents its equivalence class. Our states are then the equivalence classes found thus far, and the start state is just the equivalence class of the empty string. To determine transitions, we can simply check that if we are in the equivalence class of, say, $w \in S$, then transitioning on the letter a should lead us to the equivalence class (i.e., the state) of $w \cdot a \in S \cdot \Sigma$. Finally, the accepting states are the ones that, on the empty continuation ($\varepsilon \in E$), accept the given access strings of their class. Formally, the rows s where $T(s)(\varepsilon) = \mathcal{L}(s \cdot \varepsilon) = \mathcal{L}(s) = \top$.

With this definition in mind, we can see why there are certain conditions needed on the observation table for an automaton to be extracted. For this definition to be well-defined we need naturally that $\varepsilon \in S$, so that $q_0 \triangleq T(\varepsilon)$ is indeed a state of the automaton, and that $\varepsilon \in E$ too, as used by the definition of F. This condition is easy to maintain: let us define the initial observation table such that $S = E = \{\varepsilon\}$.

The more interesting conditions though, are on ensuring the transition function is well-defined. In particular we must have that $T(w \cdot a)$ is defined, for every $w \in S$.

This is ensured by the $S \cdot \Sigma$ rows of the table. More importantly, however, we must have that $T(w \cdot a) \in Q$, i.e., that there exists a corresponding row in the S portion of the table. We call this condition closedness.

Definition 2.3.4 (Closed Table)**.** *An observation table T is closed if:*

$$\forall s \in S. \forall a \in \Sigma. \exists s' \in S.\ T(s \cdot a) = T(s')$$

△

To ensure we maintain this condition, whenever we detect a $s \cdot a \in S \cdot \Sigma$ that does not have some corresponding s row, we can simply promote this row: $S \leftarrow S \cup \{s \cdot a\}$.

The last condition is important in ensuring the transition function is *deterministic*. As the observation table grows, it is entirely possible that there are more access strings $s \in S$ than there are unique corresponding rows. I.e., there could be more than one access string found for (allegedly) the same equivalence class / state. This is not problematic, as long as their extensions are *consistent*. That is to say, if s and s' are allegedly access strings of the same state, then by determinism, they should transition to the same states, too. This condition is called consistency.

Definition 2.3.5 (Consistent Table)**.** *An observation table T is consistent if:*

$$\forall s, s' \in S. \forall a \in \Sigma.\ T(s) = T(s') \implies T(s \cdot a) = T(s' \cdot a)$$

△

Failures in consistency are a blessing in disguise. Whenever we have an inconsistency, there exist then two access strings s, s' that allegedly represent the same equivalence class / state ($T(s) = T(s')$), however on transitioning on a they go into different equivalence classes / states ($T(s \cdot a) \neq T(s' \cdot a)$). However, this behavior is not possible in DFAs. It must then be that what we assume to be access strings of the same class, are actually access strings of different classes / states, that simply do not yet have a distinguishing suffix in E witnessing their difference. I.e., their rows are not yet *long enough* to cover the portions where they would not be equal. To fix

this, we extend E with a new suffix. We know already as $T(s \cdot a) \neq T(s' \cdot a)$, there exists at least one cell of the row where they differ. I.e., there exists an $e \in E$ such that $T(s \cdot a)(e) \neq T(s' \cdot a)(e)$. We can use this suffix to then obtain a new, longer one: $E \leftarrow E \cup \{a \cdot e\}$. Thus distinguishing our access strings, and creating a new row. This gives birth to the discovery of a new Myhill–Nerode equivalence class, and thus a new state of the canonical automaton.

2.3.3 The L* Algorithm

Algorithm 2.1: The L* Algorithm

Data: Alphabet Σ, membership oracle $mem_{\mathcal{L}}$, and equivalence oracle $eq_{\mathcal{L}}$.
Result: The learned DFA $\mathcal{A}$ accepting $\mathcal{L}$.

$S, E \leftarrow \{\varepsilon\}$; // Initialize empty observation table.
$cex \leftarrow \varepsilon$;
while $cex \in \Sigma^\star$ **do**
  $S \leftarrow S \cup \mathsf{prefixes}(cex)$; // Extend observation table.
  for $s \in S, a \in \Sigma \cup \{\varepsilon\}, e \in E$ **do** // Fill in new cells.
    $\mathsf{row}(s \cdot a)(e) \leftarrow mq_{\mathcal{L}}(s \cdot a \cdot e)$;
  if $\exists t \in S \cdot \Sigma.\ \forall s \in S.\ \mathsf{row}(t) \neq \mathsf{row}(s)$ **then**
    $S \leftarrow S \cup \{t\}$; // Enforce closedness (Definition 2.3.4).
    continue;
  if $\exists s_1, s_2 \in S, a \in \Sigma, e \in E.$
   $\mathsf{row}(s_1) = \mathsf{row}(s_2) \wedge \mathsf{row}(s_1 \cdot a)(e) \neq \mathsf{row}(s_2 \cdot a)(e)$ **then**
    $E \leftarrow E \cup \{a \cdot e\}$; // Enforce consistency (Definition 2.3.5).
    continue;
  $q_0 \leftarrow \mathsf{row}(\varepsilon)$; $Q \leftarrow \{\mathsf{row}(s) \mid s \in S\}$; // Build automaton (Definition 2.3.3).
  $F \leftarrow \{\mathsf{row}(s) \mid s \in S, \mathsf{row}(s)(\varepsilon) = \top\}$;
  $\delta(row(s \in S), a \in \Sigma) \leftarrow row(s \cdot a)$;
  $\mathcal{A} \leftarrow (Q, q_0, \Sigma, \delta, F)$;
  $cex \leftarrow eq_{\mathcal{L}}(\mathcal{A})$; // Query equivalence oracle for counterexample.
return $\mathcal{A}$;

The L* algorithm (Algorithm 2.1) then proceeds in this iterative fashion: (1) The current observation table is made closed and consistent, as described above, and its cells completed with membership queries to the teacher. (2) A hypothesis automaton is extracted from the observation table. (3) An equivalence query is made for the hypothesis. If the hypothesis is correct we terminate, if not, we add the returned counterexample, and its prefixes, to S, and we go back to (1).

This process is guaranteed to make progress and terminate: Intuitively, each

counterexample leads to at least one new equivalence class / state being found. As the Myhill–Nerode theorem guarantees a finite number of equivalence classes, the $L^\star$ algorithm will make at most as many equivalence queries as there are equivalence classes, before returning the canonical DFA for the hidden language $\mathcal{L}$. Furthermore, Angluin shows that if the target canonical DFA has n states, and the longest counterexample returned by the teacher has length m, then the learning process will require a number of membership queries polynomial on n and m.

Lemma 2.3.6 (Correctness & Termination [Ang87, Theorem 6])**.** *Given an alphabet Σ, and an MAT teacher $\langle mq_{\mathcal{L}}, eq_{\mathcal{L}} \rangle$ for a regular language $\mathcal{L}\colon \Sigma^\star \to \mathbb{2}$, the $L^\star$ algorithm (Algorithm 2.1) terminates and returns the canonical DFA $\mathcal{A}_{\mathcal{L}}$.*

The beauty of $L^\star$ is in its simplicity: It operationalizes the Myhill–Nerode theorem by iteratively refining the Myhill–Nerode relation. The construction of the automaton is direct from the definition of the canonical DFA. As such the only way it can be temporarily wrong is if indeed it has not yet refined the Myhill–Nerode relation enough. Each refinement then finds at least one new equivalence class (i.e., a new state), a process which must terminate given the finiteness of relation.

2.3.4 Other Learning Algorithms

The $L^\star$ algorithm powered by the Myhill–Nerode relation forms then a cornerstone of automata learning. Not only due to being one of the first active learning algorithms, but because refinement of the Myhill–Nerode relation is still how the vast majority of automata learning algorithms operate. Further algorithms such as KV [KV94], TTT [IHS14], and even the more recent $L^\sharp$ [Vaa+22] still use the same MAT framework to learn canonical automata through Myhill–Nerode.

KV made significant improvements to the internal data structure used by the learner. Instead of an observation table, it uses a *classification tree*. The idea is to notice that we do not really need to have full "row" information for each access string representing the equivalence classes, rather we need at most one specific suffix per pair of access strings, that indeed distinguishes the strings as two different equivalence classes. This data structure makes more efficient use of queries, and does

away entirely with having to maintain closedness and consistency, which was itself already remedied by improvements due to Rivest and Schapire [RS93]. We will study classifcation trees in more detail as part of the contributions of Chapter 4. TTT takes this further by making improvements to counterexample handling, achieving optimal linear space complexity. Finally, the $L^\sharp$ algorithm turns the tables: it considers *apartness*, a constructive version of inequality, to distinguish strings in different Myhill–Nerode relations, and uses an *observation tree* as the data structure. This structure can be seen as a more automata-centric way of building the Myhill–Nerode relation, making extracting its corresponding hypothesis a more direct process.

The ramifications of $L^\star$ do not stop with more efficient algorithms: The reader may have noticed that $L^\star$ actually has very little to do with the specifics of DFAs. This connection is done through intermediate Myhill–Nerode properties, such as the finiteness of the equivalence relation, and the connection to a canonical automaton. As we saw in Section 2.2.1, one can easily adapt the guarantees of Myhill–Nerode to transducers such as Mealy machines. By changing the language and automaton type of the minimally adequate teacher, we gain very direct adaptations of $L^\star$ to Mealy machines [Mar+04; SG09; vHee14], and algorithms such as KV, TTT, $L^\sharp$ are capable of learning Mealy machines through similar mechanisms.

One might then wonder which other types of automata admit Myhill–Nerode style characterizations, and give us natural automata learning algorithms? This question can be answered positively for many interesting types of automata, such as symbolic automata [DD17] that are able to model languages over very large, or even infinite alphabets, and nominal automata [Moe+17] that are able to capture some languages that require an infinite number of states. However, we quickly start hitting some limitations of extending Myhill–Nerode the further we want to go from DFAs. This is not surprising, the beauty of Myhill–Nerode is precisely its connection to the inner workings of DFAs.

For example, already attempting to learn *non-deterministic* finite automata (NFAs) in a Myhill–Nerode fashion gives us some issues: state-of-the-art algorithms such as $NL^\star$ [Bol+09] lose guarantees such as state minimality. NFAs are just as

expressive as DFAs, however their non-determinism allows them to be (sometimes exponentially) more compact. They are also deceivingly simple, as they actually coincide with weighted automata over Boolean weights. Some learning algorithms for weighted automata such as $\mathrm{WL}^\star$ [Van+20] further generalize the $\mathrm{NL}^\star$ algorithm to other types of weights, and incur similar limitations such as non-minimality, and sometimes non-termination. We study this line of work further in Chapter 6, with an algorithm for learning minimal weighted automata that does not rely on Myhill–Nerode generalizations. Other generalizations of Myhill–Nerode have been made to learn subclasses of richer automata, such as register automata [Cas+16; Die+24], and timed automata [HJM20; VEB23; Bru+26].

In this section we provided a key overview of the theory backing many central active automata learning algorithms, as needed for the contributions of this thesis. Further nuanced details of automata learning algorithms and their applications will be presented as and where needed, providing a more linear reading experience. The field is naturally, extremely broad and rich, spanning more than 4 decades and enjoying many different approaches. For an overview of *grammatical inference*, the parent field of automata learning, we point the reader to the works of De La Higuera [De 10] and Wieczorek [Wie17]. For a recent survey on further algorithms and applications of automata learning we refer the reader to the work of Ali et al. [ASZ21]. Finally, we refer the reader to the research agenda of Fortz et al. [For+26] for a window into the future of automata learning, and obstacles that remain.

Chapter 3

Closed-Box Analysis of Network Protocol Implementations

In this chapter we present PROGNOSIS, a *modular* and *reusable* framework for learning and analyzing models of network protocol implementations, specializing in bug finding and knowledge acquisition.

Implementations of network and security protocols such as TCP/IP, SSH, DTLS, and QUIC are prominent components of many internet and server applications. Errors in these implementations are common causes of security breaches and network failures which can cause massive disruptions to important services.

Automata learning techniques have been employed to learn models, and detect bugs in implementations of network protocols, starting with the TCP protocol [FJV16]. However, these approaches are *monolithic*: They require one to *manually* design a mapper between the *abstract* traces of the model and the *concrete* traces of the implementation, a task requiring expert knowledge of the protocol logic. This task alone has been central to extending previous work to other protocols, such as SSH [Fit+17] and DTLS [Fit+20]. Instead, we consider the following question:

Can we design general and reusable techniques to detect logic errors affecting the interaction with protocol implementations without knowing a priori what properties these errors may violate?

Unlike existing approaches, PROGNOSIS can easily be adapted to handle different protocols and protocol implementations, and the programmer *does not* need to

manually program the logic to map abstract traces to concrete traces of the protocol implementation, relying instead on a *reference implementation* that they trust by instrumenting its protocol logic components. Although any valid implementation of the protocol can be used as a reference implementation, different implementations will vary in their ease of modification depending on how they were designed. The PROGNOSIS Adapter, generated from the instrumentation, creates the ideal separation of concerns between implementation-specific details and protocol-specific details. Crucially, the same Adapter can be reused for different implementations of the same protocol with a simple change, as reference implementations are, by design, able to communicate with any other valid implementation of the same protocol. Furthermore, PROGNOSIS makes it easy to experiment with different levels of model abstraction and with several analyzes that expose varying types of bugs.

We use PROGNOSIS to analyze 1 implementation of TCP and 4 of QUIC: Quiche (Cloudflare), Google QUIC, Facebook mvfst, and QUIC-Tracker [PDB18]. For QUIC, we experiment with varying levels of abstraction and learn models that: 1) only characterize the packet's frames, and 2) contain concrete data such as packet numbers. The first type of abstraction allows us to use existing decision procedures to determine whether the models learned from different implementations are equivalent. When dealing with protocols as big as QUIC for example, finding a difference in the models does not necessarily indicate a bug, it can also signal different design decisions as allowed by QUIC's flexible specification. Nonetheless, some of these differences the models capture provide greater insight into the consequences of specific design decisions that are sometimes taken lightly, or sections of the specification that should be stricter. We found one such case which triggered a change to the specification to better formalize the intended behavior. Abstract models also allow us to verify and test safety and liveness properties. By inspecting counterexamples produced by checking safety properties we identified 3 bugs in QUIC implementations, which have been acknowledged by the developers.

The remainder of the chapter is organized as follows:

In Section 3.1 we present PROGNOSIS's modular architecture that enables

reusability: different protocols and protocol implementations can easily be swapped without changes to the learning engine.

In Section 3.2 we detail the *System under Learning* (SUL) module, comprised of the Implementation being learned, and the Adapter component that enables configurable levels of abstraction in learned models. We show how, unlike with adapters used in previous approaches, we can build an Adapter using a reference implementation, saving programmers time and the need for domain expertise.

In Section 3.3 we describe the Learner module, and how it uses automata learning algorithms to produce base models of the SUL via interaction. Through program synthesis, these models are then *enriched* with more expressive properties, such as patterns in data flow throughout the execution of the protocol. Synthesizing these properties *a posteriori* helps with scalability and performance yet does not compromise correctness. Once a bug is discovered at the model level, PROGNOSIS creates concrete traces to check whether the bug is in the implementation or is a false positive (that can be used to refine the model).

In Section 3.4 the analysis module is presented, which allows PROGNOSIS to detect bugs in closed-box implementations, even in the absence of specifications. This is achieved through a range of model-based testing and decision procedures.

In Section 3.5 we detail the results of the orchestration of these modules: Case studies over a TCP implementation and several QUIC implementations. The TCP case study shows that our framework can recover previous results [FJV16], albeit with much less manual configuration. The QUIC case study highlights that the framework can be used to uncover bugs as well as to gain insight into different design choices and aid in RFC improvements. One of the bugs (if exploited) could make the QUIC server vulnerable to Denial-of-Service attacks and it highlighted that in fact, a part of the protocol was totally missing from the implementation.

We review related work in Section 3.6 and conclude in Section 3.7 with a discussion of the limitations of the framework, and directions for further work.

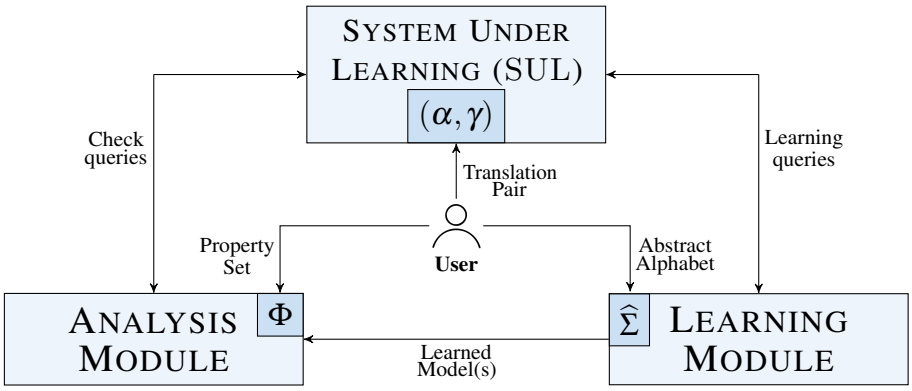


Figure 3.1: PROGNOSIS' modular architecture.

3.1 Architecture

In this section, we give an overview of the architecture of our framework. PROGNOSIS is comprised of three modules (see Figure 3.1), which are parameterized by inputs provided by the user who is analyzing the protocol.

3.1.1 System Under Learning (SUL)

The SUL is the protocol implementation we are analyzing, for example, a TCP server. This implementation is accessed in a closed-box fashion: All we assume is that we can send packets to and receive packets from it. In general, implementations are complex, and we cannot hope to directly learn models of their entire behaviors. A user of PROGNOSIS provides an Adapter translation pair (α, γ) containing an abstraction function α that maps concrete packet traces of the SUL to simplified abstract traces. For example, the function α might simplify TCP packets to only consider whether they are of type `SYN`, `ACK`, or `SYN-ACK`. We also need a concretization function γ that maps simplified abstract traces to concrete ones accepted by the SUL. Unlike existing approaches that require the user to explicitly implement this function [FJV16], PROGNOSIS only requires the user to instrument an existing reference implementation of the same protocol to produce concrete packet traces from abstract ones. We describe this setup in detail in Section 3.2. With an Adapter that can map between abstract and concrete traces, we have the interface required to learn models and perform analysis.

3.1.2 Learning Module

We use existing active automata learning algorithms [FHS25] to learn a model of the SUL. Active learning algorithms work by directly interacting with the SUL instead of relying on passive data such as logged events or historic data. This allows us to do closed-box learning, where we don't have access to the inner workings of the SUL, and guarantees that the learner is able to get answers to the queries it makes.

Specifically, we focus on algorithms based on the MAT framework introduced in Definition 2.3.1, where the learning algorithm obtains its information through queries. On an abstract level, the models operate over the simplified abstract traces and therefore abstract away much of the complexity of the SUL, allowing the user to focus on the properties of interest.

On a concrete level this involves sending sequences of input packets to the SUL, and registering the response obtained. For example, the learner might ask what happens when a `SYN` packet is sent and followed by an `ACK`. The SUL uses its Adapter to, acting as a client, produce and send these packets to the implementation, and return an answer the learner uses to build an accurate model.

In Section 3.3, we show how PROGNOSIS can learn different models of varying levels of abstraction, from plain deterministic models to models that include registers to store e.g., packet numbers.

As a central example, we will show how PROGNOSIS can be used to learn a detailed model of the TCP 3-way handshake shown in Figure 3.3.

3.1.3 Analysis Module

This module enables the use of the learned models to analyze the SUL, using a portfolio of techniques to unveil complex bugs and help the user gain insights about the implementation's behavior. For example, PROGNOSIS can automatically determine whether the models learned for two different implementations are equivalent (for different notions of equivalence) and also supports simple visualizations of the learned models that allow a user to visually compare two models for differing behaviors. In Section 3.5, we show how the automated equivalence check and visualizations aided in detecting anomalies and explaining such anomalies to real developers in

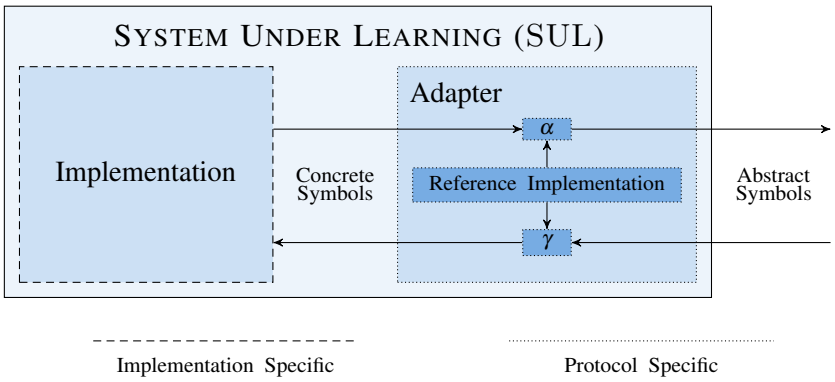


Figure 3.2: System Under Learning components.

our evaluation. For example, PROGNOSIS could detect that a supposedly variable value being transmitted was actually a constant. The full set of analysis exposed by PROGNOSIS is discussed in Section 3.4.

3.2 System Under Learning (SUL)

The SUL has two sub-components: an Implementation we want to analyze and model, and a protocol-specific Adapter that uses a translation pair to transform concrete packet traces into simplified abstract traces, and *vice versa*. The Adapter is the interface with the learning module (Figure 3.2). In fact, the learner is oblivious to the existence of concrete traces, and communicates only with the Adapter, which then interacts with the Implementation. The learning module (described in Section 3.3) then uses the abstract traces to build models of the Implementation.

3.2.1 Implementation

The Implementation is the original system we are attempting to learn, for example a running TCP server. In this case we host the implementation being learned, but simply being able to connect to the implementation is enough for PROGNOSIS to learn a model. At the high level, the implementation takes sequences of inputs in a domain Σ and produces sequences of outputs in a domain Γ.

In what follows, we refer to these domains as *alphabets*, the usual terminology in the automata learning literature. We will be using different alphabets that get gradually more abstract: the native alphabet, the concrete alphabet, and the abstract

alphabet. Each of these alphabets serves its own purpose, and they are the way in which the PROGNOSIS modules communicate. In general, the input and output alphabets of the implementations are complex packets. The following example shows the low-level alphabet used in typical network communications.

Example 3.2.1 (Native Alphabet)**.** *The* native alphabet *of a TCP implementation consists of all the possible TCP packets in their binary format, with all their complex fields and restrictions. The input and output alphabets are both the binary representations of TCP packets, formally binary sequences ($\mathbb{2}^\star$). These sequences correspond to the packets that will be sent over the wire to the* Implementation *and received back as a response. For example, in our analysis of the TCP* Implementation *in Figure 3.3, we use the* Scapy *[Bio03] Python library to send arbitrary TCP packets in $\mathbb{2}^\star$.*

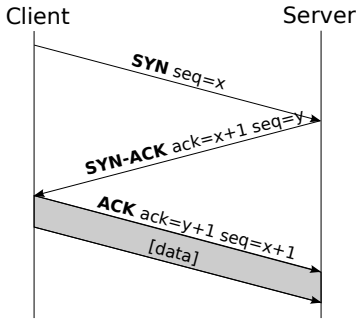
The native alphabet is only useful for creating packets to be communicated, but in general, it is more convenient to use structured alphabets that capture the fields and values of packets. Even though packets are communicated in binary, these same can be represented as a structured alphabet, for example, as a *JSON* object.

Example 3.2.2 (Concrete Alphabet)**.** *In our TCP example, the concrete alphabets can be JSON objects with the following structure:*

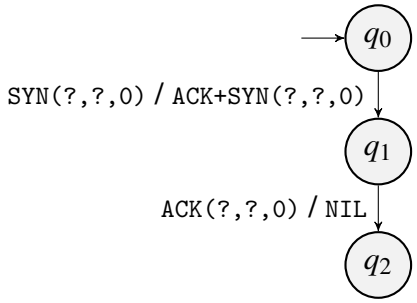
```
{ "isNull": false,
  "sourcePort": 40965,
  "destinationPort": 44344,
  "seqNumber": 48108,
  "ackNumber": 0,
  "dataOffset": null,
  "reserved": 0,
  "flags": "S",
  "window": 8192,
  "checksum": null,
  "urgentPointer": 0 }
```

We will use concrete alphabets, which we below refer to as Σ and Γ, as a machine-readable representation of the native alphabets. Concrete alphabets and native alphabets are necessary to trigger concrete executions of the SUL. However, these alphabets are simply too large (sometimes infinite). When learning a model

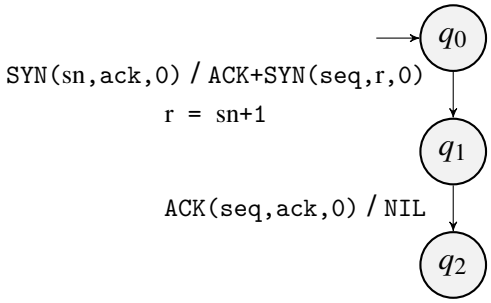
of an implementation, one has to abstract some parts of the alphabets, such as specific packet formats and encrypted messages, to make automata learning feasible and helpful. For example, while it is not feasible to learn a model describing an exact TCP implementation, it is possible to learn a model of what types (e.g., `SYN`, `ACK`, `SYN-ACK`) of packets are exchanged during a handshake (Figure 3.3b). This abstraction is handled by the Adapter, as described next.



(a) The TCP 3-way handshake [LU13].



(b) Learned model (fragment).



(c) Synthesized model (fragment) with register r.

Figure 3.3: TCP handshake: protocol, learned model, and synthesized model.

3.2.2 Adapter

In the architecture of our system (Figure 3.1), the Adapter, which is part of the SUL (Figure 3.2), is by far the hardest component the user needs to provide. The Implementation itself is provided to us, and the Learner is general, but the Adapter has to translate inputs that the Implementation understands to inputs that the Learner can use, and *vice versa*. The Adapter has to know the logic of the protocol to produce concrete packets with the right parameters, and it needs to know how to encode concrete symbols into their binary representation to interact with the Implementation, acting as the client in the connection. Instead of requiring the user to provide an

implementation of the Adapter from scratch, PROGNOSIS lets the user resort to a *reference implementation* as basis to build the Adapter. Before we describe this key idea, we set some notation.

The Adapter translates concrete input/output traces into simplified traces for which it is feasible to do automata learning. We call the latter abstract traces as they are built from *abstract alphabets* $\widehat{\Sigma}$ and $\widehat{\Gamma}$, allowing us to abstract away details that would make the alphabet otherwise too big or complex. Let us now illustrate a candidate abstract alphabet to learn the TCP handshake of Figure 3.3.

Example 3.2.3 (Abstract Alphabet)**.** *In Figure 3.3a we have packet flags, sequence, and acknowledgment numbers. Because we are only interested in modeling what types of packets are exchanged, our abstract alphabets* $\widehat{\Sigma}$ *and* $\widehat{\Gamma}$ *will only contain packet flags. For example, a packet might have the following structure:* `ACK+SYN(?,?,0)`*. Here, each* `?` *represents the sequence and ack number parameters left unspecified. While it seems unnecessary to have these parameters if they are not used in learning, we will use them in Section 3.3.2 to synthesize a richer model.*

Remark 3.2.4 (Nondeterminism)**.** *The abstract trace in the previous example does not incorporate the* `seqNumber`*. Because in TCP the Sequence Number is randomly determined at the start of the connection, we could have two different traces that represent the same 3-way TCP handshake. A choice the user has to make when providing an abstract alphabet is what they plan to model. In this example, we assume the user is trying to learn a model of the 3-way handshake and they know this process should be deterministic. Therefore, providing elements of the abstract alphabet that will cause nondeterminism would be a poor choice of abstract alphabet.*

Section 3.5 shows how PROGNOSIS *provides mechanisms to detect when a choice of abstract alphabet results in nondeterminism. In Section 3.5.6, we show a case in which nondeterminism was the result of an undesired protocol behavior.*

Going from a native alphabet, to a concrete alphabet, to an abstract alphabet is somewhat simple as we merely remove information deemed unnecessary for learning. However, these details are essential for the learner to communicate with the

Implementation, so this missing information will need to be recovered when sending concrete packets to the SUL. Learning algorithms (and our analysis) often need to perform queries to the SUL to decide how the model should be constructed. These queries, given an abstract trace a, require one to construct a concrete trace c that is valid in the Implementation and that corresponds to a. This process is handled by the translation pair of the Adapter.

Definition 3.2.5 (Translation Pair)**.** *A* translation pair (α, γ) *is formed of an* abstraction function α *that maps pairs of input/output traces to abstract traces; and a* concretization function γ *satisfying* $\alpha(\gamma(a)) = a$*, where:*

$$\alpha\colon \Sigma^\star \times \Gamma^\star \to \widehat{\Sigma}^\star \times \widehat{\Gamma}^\star \qquad \gamma\colon \hat{\Sigma}^\star \times \hat{\Gamma}^\star \to \Sigma^\star \times \Gamma^\star$$

△

Example 3.2.6. *As an example of an abstraction function, we could be masking all fields except the sequence and ack number, providing a* payload length *instead of the actual payload, and unspecifying the* `seq` *and* `ack` *numbers of TCP Packets: taking* {..., `flags: SYN, seq: 123, ack: 55`} *to an abstract TCP Packet* {`flags: SYN, seq: ?, ack: ?, payloadLen: 0`}*. Notationally, we write these abstract symbols compactly as* `SYN(?,?,0)`.

Designing the reverse process, that is the concretization function, is a hard task that requires expert knowledge and stands in the way of modularity and reusability. Simplifying this problem is one of the aspects where our work significantly differs from prior work: Fiterău-Broştean et al. [FJV16] have used an architecture reminiscent of ours to learn models of TCP implementations, but require the user to provide a Mapper, which is effectively an implementation of the concretization function γ. Directly implementing γ requires a user to know the protocol logic in detail and to understand what concrete packets are valid and not valid. Essentially, the user needs to implement part of the protocol itself, which is a hard, laborious task that significantly impacts the cost-benefit of automata learning. Moreover, this explicit implementation of a concretization function is close to impossible if the protocol

relies on a logic of high complexity (as is the case for QUIC), including aspects like key derivation, encryption, or symbols that contain a large number of fields.

Our solution to implementing concretization functions is inspired by the following common expression in cryptography: *Never roll your own Crypto*. I.e., because cryptographic algorithms tend to be of great complexity, it is best to use existing implementations that have been widely tested instead of implementing them yourself. Building on this insight, PROGNOSIS exploits the following key idea.

3.2.2.1 Reference Implementation as a Concretization Oracle

Instead of manually implementing a concretization function from scratch (i.e., a version of the protocol logic that can produce concrete traces from abstract ones), we rely on a given *reference implementation* to provide ground truth information to the Adapter. Concretely, we rely on the reference implementation to both do the concretization logic and native formatting, as a normal implementation would. Given an abstract query symbol a the Adapter needs to find a concrete packet c to build that matches a. We modify the reference implementation so that it can "abstractly execute" abstract packets to identify what concrete packets they can yield. While this sounds like a mouthful, it boils down to identifying and using the parts of the reference implementation that operate over the symbols we are interested in.

The main advantage of using a well-tested reference implementation is that it already encodes the desired protocol logic: In what order packets are transmitted, how they are built, etc. For example, a natural choice for the QUIC reference implementation is QUIC-tracker [PDB18], which is widely used by QUIC implementers.

Note that this implementation does not have to be 100% correct. It is merely a reference point for modeling the interaction with the target implementation. While correct reference implementations allow us to narrow detected bugs to the target implementation, finding any bug is useful as it detects issues in either implementation.

Naturally, although a reference implementation allows us to communicate with the target implementation, it also has behavior that is not fit for being modeled. Namely, via our modification of the reference implementation we aim to enforce the following properties:

(1) No unrequested packets are sent to the target Implementation. We must ensure that any output symbols registered are indeed caused only by the input packets requested by the Learner's query.

(2) All concrete packets sent match the requested abstract packets. Concrete packets must fulfill the requested abstract symbols fully. Put more formally, when abstracted by α they equal the requested abstract symbol.

(3) Both reference and target implementations can be reset on request, returning them to their initial state. The learner must be able to perform a series of independent queries, as such we must have a way to fully reset the connection to its initial state, ready for a new query to be done.

(4) Concrete packets constructed or received as a response are saved with their abstract counterparts in a historic Oracle Table. The Oracle Table is a critical data structure used in synthesizing more detailed models of the implementation. It is further explored in Section 3.3.2.

(5) Response packets from the target implementation to the reference implementation are abstracted and sent back to the learner. Just as the adapter must be able to concretize packets, it must be able to abstract them back to the same abstraction level as the original request it received, so that they can be sent back to the learner.

Although the specific ways of achieving these properties vary from an implementation to another, we have found this requirement set is often enough to strike the balance in allowing the learner to interact with the SUL while minimizing behavior introduced by the reference implementation.

In some more complex protocols, it may also be useful to store in a queue formulated payloads that would break rule (1) by reacting to a received response packet, so that they can be sent later if requested by a matching abstract counterpart, as exemplified in Listing 3.1.

Our modified code will have instrumentations at all the points where packet types and packet frames are constructed, and ensure through the instrumented condi-

tional branching that only matching concrete packets are sent to the target implementation. We also add code to be able to hook into functionality allowing us to create packets from scratch.

As an example, consider the Adapter receives an abstract query `INITIAL[ACK]` for a QUIC packet (where `INITIAL` is one of the 6 packet types and `ACK` is one of the over 30 frames carried in packets).

Firstly, the queue is searched for a matching packet to be sent. If there is no match for this packet, one is made from scratch, with the valid current state of the reference implementation (Packet Number, connection details, and others), and sent instead. The reference implementation may then receive packets as a response, which are processed by the reference implementation as usual to make sure its state is updated, and if the reference implementation logic would cause a new packet to be sent in response, it is instead stored in our queue waiting for the learner, as demonstrated below.

Listing 3.1: Enforcing (1) and (2) in ACK sending logic

```
  if (packet.shouldBeAcked) {
    // Received a packet to be acked
-   connection.sendPacket(INITIAL, ACK)
+   AckQueue += packet
  }
+ if (requestedInitialAck) {
+   // Abstract Symbol was requested
+   connection.sendPacket(AckQueue.find(INITIAL, ACK))
+ }
```

After this, the original response from the target implementation is abstracted, and sent to the learner:

Listing 3.2: Implementing (4) and (5) in packet send/receive hooks

```
+ // Define abstract symbol according to set abstraction.
+ abstractSymbol = newAbstractSymbol(
+   packetType, version, packetNumber, frameTypes)
  (...)
+ // Save symbol exchange in Oracle Table.
+ oracleTable.addIOs(abstractInputs, abstractOutputs,
+                    concreteInputs, concreteOutputs)
+ // Return abstract response to learner.
```

```
+ learner.send(abstractOutputs)
```

Finally, at the end of each query the Learner requests that the SUL reset its state for a new query to be made. In the specific case of QUIC, it is enough to reset the Adapter (client), and start a new connection to the Implementation:

Listing 3.3: Implementing (3) in main Adapter body

```
+ // Define reset routine.
+ func reset() {
+   agents.stopAll()
+   connection.close()
+   connection = newConnection(conncetion.getConfig())
+   agents = defaultAgentsWithConnection(connection)
+ }
```

Most importantly, by using a reference implementation we can rely on it to maintain the desired state between multiple packets: When a new abstract symbol request comes in, we can simply resume from the state we left at the previous abstract symbol. This last aspect is what allows us to not have to explicitly model any of the protocol-specific logic, as it was done in prior work [FJV16].

To quantify this aspect, the mapper implemented by Fiterău-Broştean et al. [FJV16] to model the concretization function consisted of 2,700 lines of code implementing the logic of TCP. For QUIC, a protocol that replaces not only TCP entirely, but also TLS (encryption), and HTTP (application layer), our instrumentation required just over 2,000 lines of mostly boilerplate code, enforcing the properties listed above. Given the complexity of QUIC (which involve cryptography and other complex features), it is hard to imagine replicating the approach from [FJV16].

To make the approach even more effective, we introduce a few optimizations. First, whenever the reference implementation sends new packets (as part of an intermediate step) that do not match our abstract query, we store such packets in a list and use them to try to answer future queries. That is, when an abstract query comes in, we first look into this list to check whether any of the packets has the desired abstract value and, if this is the case, we send that packet to the Implementation. Second, we cache all intermediate pairs between abstract I/Os and concrete I/Os in a data structure called an Oracle Table.

Definition 3.2.7 (Oracle Table)**.** *An* oracle table *O is a set of corresponding abstract / concrete pairs. Formally:* $O \subseteq \{(a,c) \mid a \in (\widehat{\Sigma}^\star \times \widehat{\Gamma}^\star), c \in (\Sigma^\star \times \Gamma^\star)\}$ △

In Section 3.3, we use this cache to synthesize richer models that go beyond finite abstract alphabets and can capture concrete packet numbers and other numerical quantities. We keep track of received packets for each query and also use this record to detect retransmitted packets that should not be relayed due to nondeterminism.

Now that we have constructed an abstraction that produces a simplified abstract alphabet, and we have effective ways of querying over this alphabet, we have built the interface necessary to interact with the SUL module. We could then, for example perform our 3-way TCP handshake by sending the input trace `SYN(?,?,0)` `ACK(?,?,0)` and we would get the output trace `ACK+SYN(?,?,0)` `NIL`, which accurately represents the flags of our 3-way TCP handshake. However, because it uses our abstract alphabet the Adapter cannot tell us the exact sequence or ack numbers it picked as this would cause nondeterminism.

3.3 Learning Module

We now describe how the learning module interacts with the SUL to learn models of the implementation. In Section 3.3.1, we recall the queries the learner is allowed to ask to construct a model, and recall existing techniques for learning Mealy machines in practice. In Section 3.3.2 we use program synthesis to learn a more detailed model that is capable of recovering register values and changes, like sequence and acknowledgment numbers in the TCP protocol.

3.3.1 Learner Interface

Thanks to the techniques presented in Section 3.2, the SUL can be treated as a *query oracle* that can answer the question "*If I send this input sequence, what will the implementation return?*". With this deterministic query oracle we have all we need to fulfill the role of a Minimally Adequate Teacher (see Definition 2.3.1) for the behavior of the SUL.

To fulfill the need for membership queries ($mq\colon \widehat{\Sigma}^\star \to \widehat{\Gamma}^\star$), whenever the Learner asks for the behavior of an input $a_I \in \widehat{\Sigma}^\star$, we can simply request this trace through

the Adapter's interactions, and return its response $a_O \in \widehat{\Gamma}^\star$ to the Learner.

In practice, equivalence queries that inform the Learner of potential counterexamples to a model require an oracle omniscient of the SUL, and if we had that, we would not have to learn the system in the first place. Instead, we can use heuristic Equivalence Oracles such that when a counterexample is returned, it is guaranteed to be a valid counterexample, but the absence of a counterexample no longer guarantees equivalence. This approach still gives us approximation guarantees: i.e., the model is accurate with high probability with respect to the set of inputs we use to test equivalence. It is now a good point to remind the reader that, in the spirit of lightweight formal methods, the goal of this framework is to unveil potential incorrect behaviors of the SUL, rather than provide completeness guarantees. Even if the learned models might not provide 100% coverage, they will still be helpful to analyze and detect anomalies as we will show in our evaluation.

PROGNOSIS then uses the TTT algorithm [IHS14] for learning Mealy machines (see Section 2.2.1), which allow us to capture the (abstract) packet input-output patterns of the Implementation.

Example 3.3.1. *The Mealy machine in Figure 3.3b is a model of the TCP 3-way handshake over input and output alphabets:*

$$\widehat{\Sigma} \triangleq \{\texttt{SYN(?,?,0)}, \texttt{ACK(?,?,0)}\} \qquad \widehat{\Gamma} \triangleq \{\texttt{ACK+SYN(?,?,0)}, \texttt{NIL}\}$$

Given the input sequence [SYN(?,?,0),ACK(?,?,0)]*, this machine outputs the sequence* [ACK+SYN(?,?,0),NIL]*. It starts in state* q_0*, when reading* SYN(?,?,0) *it transitions to state* q_1 *and outputs* ACK+SYN(?,?,0)*, and then reading* ACK(?,?,0)*, it transitions to* q_2 *and outputs* NIL *(no packet).*

3.3.2 Synthesizing Rich Models

Mealy machines can only model operations involving finite alphabets and cannot reason about numerical values, like sequence numbers. To capture the packet exchange in Figure 3.3a we need not only the TCP flags currently learned in the model depicted in Figure 3.3b but also certain quantities: Sequence and acknowledgment

numbers. While there are direct learning algorithms for certain automata models with memory [Cas+16], these do not meet the needs of this framework, as they do not support complex comparison operations and updates. Instead, in this section, we present an extension of Mealy machines that adds registers, and numerical inputs and outputs. We then show how the ideas in Section 3.3.1 can be combined with constraint-based synthesis techniques to learn these extended models.

While we could consider a wide range of enhancements to traditional automata, we limit ourselves to extending automata to read and write integer values from packets, and to increment or set to input values one of a finite number of registers $\vec{x}$. These extensions capture common features of the protocols we are interested in.

Formally, an *extended* Mealy machine is a Mealy machine endowed with extra state: in addition to the machine being in a specific state at any time, it also has a finite set of n integer registers taking a specific valuation ($\vec{x} \in \mathbb{Z}^n$). These registers can then be updated on transitioning based on their current *configuration* $(Q \times \mathbb{Z}^n)$ and instantiated k-parameter alphabet inputs $(\widehat{\Sigma} \times \mathbb{Z}^k)$, and the machine outputs in turn an instantiated alphabet output $(\widehat{\Gamma} \times \mathbb{Z}^k)$.

Definition 3.3.2 (Extended Mealy Machine)**.** *An n-register* extended *Mealy machine over k-parameter abstract alphabets $\widehat{\Sigma}, \widehat{\Gamma}$ is a tuple $\mathcal{E} = \langle Q, q_0, \widehat{\Sigma}, \widehat{\Gamma}, \delta, \lambda \rangle$ where:*

- *Q is a finite set of states, with $q_0 \in Q$ the initial state.*
- *$\delta \colon (Q \times \mathbb{Z}^n) \to (\widehat{\Sigma} \times \mathbb{Z}^k) \to (Q \times \mathbb{Z}^n)$ is the* configuration *transition function.*
- *$\lambda \colon (Q \times \mathbb{Z}^n) \to (\widehat{\Sigma} \times \mathbb{Z}^k) \to (\widehat{\Gamma} \times \mathbb{Z}^k)$ is the output function.*

The semantics of an extended Mealy machine is given by the function:

$$[\![\mathcal{E}]\!](q,\vec{x}) \colon (\widehat{\Sigma} \times \mathbb{Z}^k)^\star \to (\widehat{\Gamma} \times \mathbb{Z}^k)^\star$$

$$[\![\mathcal{E}]\!](q,\vec{x})(\varepsilon) = \varepsilon \qquad [\![\mathcal{E}]\!](q,\vec{x})((a,\vec{i}) \cdot w) = \lambda(q,\vec{x})(a,\vec{i}) \cdot [\![\mathcal{E}]\!](\delta(q,\vec{x})(a,\vec{i}))(w)$$

△

Note that our extended Mealy machines have a standard initial state, however they *do not* have a fixed initial register valuation. For our purposes this is crucial:

the initial register values correspond to usually non-deterministic values in the state of the protocol, such as random initial sequence numbers used by TCP. These values are not always observable in the learning process, and tend to be connection-specific rather than protocol-specific. As such, our models do not aim to capture such information. Rather, our extended mealy machines focus on capturing the relational patterns between input and output packets on *all* connections, which can be observed, and detail the reactive logic of the protocol.

We depict a transition of an extended Mealy machine compactly as follows:

$$p \xrightarrow[\vec{x}=\mathbf{u}(\vec{i},\vec{x})]{I(\vec{i})/O(\mathbf{o}(\vec{x}))} q$$

Informally, if the machine is in state p and reads an abstract symbol I, possibly parametric on concrete numerical values $\vec{i}$, it updates the registers $\vec{x}$ with values determined by $\mathbf{u}(\vec{i},\vec{x})$, and it outputs an abstract symbol O parametric on the values determined by $\mathbf{o}(\vec{x})$. The update function $\mathbf{u}(-)$ can take on quite complex values, but in the following we consider each register is updated with either a copy of a register, or an input value, or one of these incremented by 1. Similarly, the output function $\mathbf{o}(-)$ can, for each parameter, output the value of a register or that value plus 1.

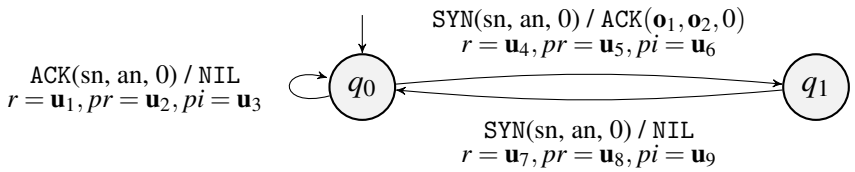


Figure 3.4: Extended machine with unknown terms.

We start from a Mealy machine where the register updates and outputs are missing (as in Figure 3.4) and our goal is to find concrete terms $\mathbf{u}_1, \cdots, \mathbf{u}_9$, and $\mathbf{o}_1, \mathbf{o}_2$ for each transition that result in an extended machine correctly modeling the SUL with respect to a given set of concrete traces.

We use the set of traces cached while learning the Mealy machine in the Oracle Table O. For each pair $(a,c) \in O$ of abstract and concrete traces, we identify what path of the extended Mealy machine is traversed by the abstract trace a and use the

concrete trace c to generate the constraints needed to identify the missing terms of each transition in the path. I.e., the terms that make the extended machine consistent with the concrete trace c. If needed, the algorithm can solicit more example traces and add them to O. The constraints are then solved using an SMT solver and the solution is used to generate the needed terms.

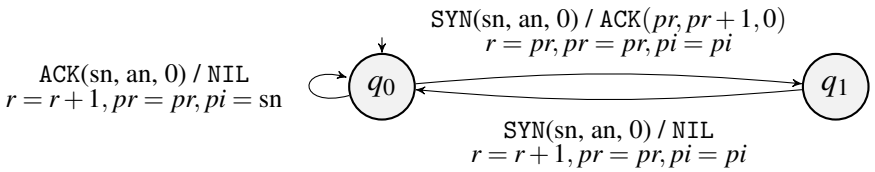


Figure 3.5: Synthesized machine over registers r, pi, and pr.

Let us illustrate how we would synthesize the extended machine in Figure 3.5 from the sketch in Figure 3.4. We consider the following concrete trace appearing in O: $[(\texttt{ACK}(0,3,0)/\texttt{NIL}), (\texttt{SYN}(2,5,0)/\texttt{ACK}(4,5,0))]$. This time, we want to model that each symbol in the concrete trace also carries the sequence number (sn) and the acknowledgment number (an). E.g., for the first input these values are 0 and 3, respectively. Given this trace, our algorithm generates constraints containing a number of variables used to denote what we are trying to synthesize and what it means for the solution to be correct with respect to the input trace. For each unknown term, we have a finite list of possible terms we can instantiate it with. For example, the unknown $\mathbf{u}_1$ can be instantiated with one of the 8 terms in the list: $[r, r+1, pr, pr+1, pi, pi+1, \text{sn}, \text{an}]$. In our constraints, we use an integer variable $E_{\mathbf{u}_1}$ to indicate the possible choices (indices start at 0). For example, $E_{\mathbf{u}_1} = 1$ indicates that the term $r+1$ will be the solution for the unknown $\mathbf{u}_1$.

As register values will change for each trace, our constraints need to model how the values of the register are updated throughout the execution. To do so, we introduce variables that track the values of each register after reading the i-th input in the trace. For example, $r[1]$ indicates the value stored in register r after reading the first input packet. When generating constraints for multiple traces, we will have a variable $r_\pi[i]$ for each trace π and index i. The following simplified set of constraints captures the synthesis problem we are trying to solve:

// Constraints for $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, $\mathbf{u}_4$, $\mathbf{u}_5$, $\mathbf{u}_6$

$0 \leq E_{\mathbf{u}_1} \leq 7$ // The value of $E_{\mathbf{u}_1}$ encodes the 8 possible terms for $\mathbf{u}_1$

$E_{\mathbf{u}_1} = 0 \implies r[1] = r[0]$ $\quad E_{\mathbf{u}_1} = 1 \implies r[1] = r[0]+1$

$E_{\mathbf{u}_1} = 2 \implies r[1] = pr[0]$ $\quad E_{\mathbf{u}_1} = 3 \implies r[1] = pr[0]+1$

$E_{\mathbf{u}_1} = 4 \implies r[1] = pi[0]$ $\quad E_{\mathbf{u}_1} = 5 \implies r[1] = pi[0]+1$

$E_{\mathbf{u}_1} = 6 \implies r[1] = 0$ $\quad E_{\mathbf{u}_1} = 7 \implies r[1] = 3$

...

// Constraints for $\mathbf{o}_1$, $\mathbf{o}_2$

$0 \leq E_{\mathbf{o}_2} \leq 5$ // The value of $E_{\mathbf{o}_2}$ encodes the 6 possible terms for $\mathbf{o}_2$

$E_{\mathbf{o}_2} = 0 \implies r[2] = 5$ $\quad E_{\mathbf{o}_2} = 1 \implies r[2]+1 = 5$

$E_{\mathbf{o}_2} = 2 \implies pr[2] = 5$ $\quad E_{\mathbf{o}_2} = 3 \implies pr[2]+1 = 5$

$E_{\mathbf{o}_2} = 4 \implies pi[2] = 5$ $\quad E_{\mathbf{o}_2} = 5 \implies pi[2]+1 = 5$

Note that setting $E_{\mathbf{u}_1} = 7$ corresponds to selecting the term `an` (i.e., the input `ACK` number) as a solution to the unknown $\mathbf{u}_1$. In this case, the constraints model that the value of the register *r* after reading the *first* packet (i.e., $r[1]$) should be equal to the `ACK` number of the *first* input packet (i.e. 3). Similarly, setting $E_{\mathbf{o}_2}$ to the value 4 corresponds to selecting the term *pi* as a solution to the unknown $\mathbf{o}_2$. In this case, the constraints model that the value of the register *r* after reading the *second* packet (i.e., $r[2]$) should be equal to the `ACK` number of the *second* output packet (i.e., 5).

Repeating this process on more examples, e.g $[(\texttt{SYN}(2,3,0)/\texttt{ACK}(4,5,0)), (\texttt{SYN}(2,3,0)/\texttt{NIL})]$, yields the automaton in Figure 3.5. This automaton corresponds to the solution that selects $E_{\mathbf{u}_1} = 1$ and $E_{\mathbf{o}_2} = 3$.

While this section shows a particular type of model, our framework is general and allows us to implement other synthesis and learning algorithms for more complex models due to its interaction with the SUL. We could, for example, allow more complex constraints on transitions, or more elaborate update functions using a mix of registers and numeric operations.

Sometimes the synthesized models can contain incorrect register patterns if the pattern is not completely covered by the Oracle Table traces. These are detected through random equivalence testing, and trigger new queries in the synthesis algorithm: The synthesizer will restart with a larger table *O* of traces, as well as negative

traces that the model should not contain. The constraints discussed in this section can be easily adapted to handle these negative examples.

3.4 Analysis Module

The last module of PROGNOSIS enables analysis techniques using the outcomes of the learning modules to help the user infer behaviors of the SUL. Different abstraction levels allow us to expose different types of anomalies; however, the analysis module is limited to uncovering logic errors captured in the model, which is restricted to observable events captured in the learning process. Some more nuanced quantity-specific bugs can be analyzed through the synthesized model.

We focus on analyzing models for bug finding and knowledge acquisition rather than providing completeness guarantees due to not all abstraction levels capturing enough information to fully verify a specification. In this section, we present some of these analyses and how they can be used to identify undesired behaviors.

3.4.1 Nondeterminism Check

As explained in Remark 3.2.4, the learning module typically expects deterministic behavior (i.e., the same input trace should trigger the same output trace). In general, nondeterminism can arise in our system for two reasons: (1) the abstract alphabet is so simplified that concrete traces corresponding to different behaviors are collapsed into the same input trace, or (2) the implementation is producing nondeterministic outputs in cases where it should not. For every Learner query, PROGNOSIS expects a deterministic answer, as it is attempting to build a deterministic model. However, due to the nature of active automata learning, environmental events such as latency and packet loss could cause nondeterminism to be observed.

To avoid this, PROGNOSIS will execute the query a specified minimum number of times, and if these are not all the same, it will continue to do so until a specified percentage of certainty is achieved. If after a limit number of queries nondeterminism is discovered, the learning process pauses, and the Adapter witnesses the cause of the nondeterminism. If reason (1) is the cause, the user will see that the abstraction is too coarse and can provide a richer abstraction. While it is possible to use more complex

learning algorithms that can handle nondeterminism in the traces, we argue that, because of reason (2), detecting sources of nondeterminism is a powerful analysis technique that can shed light on some undesired behaviors. As we show in Section 3.5.6, PROGNOSIS was able to unveil a complex bug in a QUIC implementation thanks to the nondeterminism check.

3.4.2 Learned Model Analysis

If learning succeeds, PROGNOSIS can produce visualizations of the models which can help a user understand whether an implementation works as expected. In addition, PROGNOSIS also exposes various algorithms to check properties of the learned models. For example, PROGNOSIS can test for equivalence of models of two different implementations of the same protocol (whether they accept the same input/output traces). For Mealy machines, there are algorithms for performing exact equivalence checks efficiently [HR77]. PROGNOSIS can produce concrete example traces that show the difference between the behaviors of the two implementations. We show in Section 3.5.7 how PROGNOSIS is able to unveil a complex bug in a QUIC implementation thanks to these equivalence checks, and the visualizations of the differences were instrumental in communicating the problem to the developers.

PROGNOSIS also allows the user to specify temporal properties in logics such as LTL, e.g., *Packet numbers are always increasing*, and check whether the models satisfy such properties. For Mealy machines, this procedure simply boils down to checking that the traces of the model are a subset of those allowed by the property, a decidable problem [HR77]. For extended machines with counters, this problem is in general undecidable [NSV04, Corollary 5.2] and we rely on randomized testing to make the analysis practical.

3.5 Results

We evaluate PROGNOSIS through two case studies on the following research question:

RQ1 Can PROGNOSIS be reused across different real-world network protocol implementations to learn models that expose unintended behavior?

We assess this question according to two complementary evaluation criteria. First, we assess *reusability* through the amount and location of implementation-specific adaptation required: in particular, whether the learning and analysis infrastructure remains unchanged when moving between protocols, and whether a protocol-specific Adapter can be reused across different implementations of the same protocol. Second, we assess the *analysis utility* of the resulting models through the unintended behaviors they expose, considering the number and nature of the identified issues, the analyzes through which they are detected, and, where available, external confirmation such as acknowledgment by implementation developers or changes to the protocol specification. As these criteria concern the practical reuse of the framework and the diagnostic value of the resulting models, they are assessed primarily through the evidence provided by the case studies, supplemented by quantitative measurements where informative.

The two case studies exercise these criteria in different ways. First, we apply PROGNOSIS to a TCP implementation and show that it can reproduce the level of modeling detail obtained in prior work [FJV16] with only minor adaptation, demonstrating the reusability of the framework across protocols. Second, we apply PROGNOSIS to several implementations of the IETF QUIC protocol [IT21], reusing the same protocol-specific Adapter across implementations and analyzing the resulting learned and synthesized models for unintended behavior. These case studies allow us to evaluate both the practical reusability of PROGNOSIS and whether the models it produces can support the identification of defects and specification ambiguities. All models learned and synthesized can be found in Appendix A.

PROGNOSIS was implemented using several programming languages. The general learner is implemented in 3,500 lines of Java and uses the automata learning library LearnLib [FHS25]. The extended Mealy machine synthesizer is implemented in 300 lines of Python and uses the Z3 SMT Solver [DB08].

3.5.1 Learning a TCP Implementation

In this case study, we demonstrate how PROGNOSIS can reproduce results from previous work on learning models of TCP implementations [FJV16]. We learn the

Ubuntu 20.04.1 LTS TCP stack, particularly kernel version `5.8.0-40-generic`.

We provide PROGNOSIS with the same abstract alphabet $\widehat{\Sigma}$ and translation pair (α, γ) used in prior work [FJV16]. Concretely, the abstract alphabet models the TCP flags for the packet, leaves the sequence and acknowledgment numbers unspecified, and provides the payload length:

$$\widehat{\Sigma} = \{\texttt{SYN(?,?,0)}, \texttt{SYN+ACK(?,?,0)}, \texttt{ACK(?,?,0)}, \texttt{ACK+PSH(?,?,1)},$$
$$\texttt{FIN+ACK(?,?,0)}, \texttt{RST(?,?,0)}, \texttt{ACK+RST(?,?,0)}\}$$

To implement the abstraction function α and the concretization function γ we need a reference implementation. To illustrate the flexibility of PROGNOSIS, we use the mapper implementing the TCP logic designed in [FJV16] as reference and instrument it. Note that we could have used any other TCP implementation for this task. Instrumenting the reference implementation required only 300 lines of code to integrate with *Scapy* [Bio03].

The model learned by PROGNOSIS has 6 states and 42 transitions and took 4,726 membership queries to learn. This model is slightly different to the one learned by Fiterău-Broştean et al. [FJV16] due to changes made to the implementations over time. Unfortunately, despite multiple attempts, we were not able to run the prior technique by Fiterău-Broştean et al. [FJV16] on the new TCP implementation and we cannot see what model it will learn now. This case study and the difficulty of adapting prior work to new implementations showed that we could reproduce the level of detail obtained in prior work with very minor effort thanks to the modular components of PROGNOSIS.

While we do not perform any analysis in this case study, we note that our synthesis procedure can help us derive richer models than those considered in prior work. In particular, using the technique described in Section 3.3.2, we can synthesize a model describing the behavior of extra parameters such as the `dataOffset`. Remarkably, applying this richer analysis only required changing one line of code in the implementation: simply selecting which concrete values of the TCP implementation we wish to synthesize over.

3.5.2 Learning QUIC Implementations

In this section, we analyze 4 implementations of the QUIC protocol. We devise the abstract alphabet $\widehat{\Sigma}$, provide a reference implementation for the translation pair (α, γ), and a set of properties from the QUIC specification to analyze. We show how PROGNOSIS detected behavior that led to a change in the QUIC specification, as well as identified bugs in real implementations of the QUIC protocol.

3.5.3 Background

The QUIC protocol [IT21] combines specific features of independent protocols that are commonly used together, reducing the overhead that arises from having to coordinate these features. Specifically, it collapses the services provided by HTTP, TLS, and TCP into a single *super protocol*. Therefore, QUIC is extremely complex, and its design has been carefully thought out to ensure these services that were previously isolated can be communicated efficiently. QUIC achieves this goal using encapsulation. In TCP, a single packet is enough to communicate everything needed, from specific signals like `SYN` or `ACK` to the payload of the application layer itself. In QUIC, both signaling, and data transmission happen in *frames*. A packet serves merely as a mean of safely transporting different types of frames.

In total, QUIC provides 6 packet types and 31 frame types, each responsible for signaling a specific aspect of the protocol. We will not be diving into the details here due to their complex nature, and instead, we will introduce specific frames and packets as needed to explain key properties of the protocol. We defer the reader to the QUIC specification [IT21] for further details. We will consider the following widely used QUIC implementations:

Quiche is Cloudflare's QUIC implementation [Clo18] that allows QUIC connections to any website protected by the Cloudflare network. Cloudflare's Content Delivery Network (CDN) supports over 25 million websites.

Google's QUIC implementation [Goo12] runs on Google servers and on Chromium browsers. Since October 7 of 2020, 25% of Google Chrome users had QUIC support enabled by default, with that proportion increasing over the followed weeks.

Facebook's mvfst QUIC implementation [Fac18] is used mainly in the Proxygen server implementation. This server implementation is responsible for powering most public facing connections to facebook.com, as well as API connections used by the Facebook and Instagram mobile apps.

QUIC-Tracker is an implementation [PDB18] designed to run testing scenarios over other implementations, with the goal of testing what technologies different implementations support.

Because these first three implementations account for a large portion of the web traffic, identifying erroneous behaviors in any of them is of critical importance.

3.5.4 Learning Models of QUIC Implementations

We use PROGNOSIS to learn models of the first three QUIC implementations described above and analyze a subset of the properties from IETF's Draft 29 [IT20]. These include, for example, *The sequence number on each newly-issued connection id must increase by 1* and *An endpoint must not send data on a stream at or beyond the final size*. PROGNOSIS requires 3 things from the user: An abstract input alphabet, a translation pair (α, γ) to be able to convert between abstract and concrete alphabets, and optionally, a set of properties Φ that we would like to test. We use an abstract input alphabet $\widehat{\Sigma}$ with 7 symbols:

$$\begin{aligned}\widehat{\Sigma} = \{&\texttt{INITIAL(?,?)[CRYPTO]}, \texttt{INITIAL(?,?)[ACK,HANDSHAKE_DONE]},\\ &\texttt{HANDSHAKE(?,?)[ACK,CRYPTO]},\\ &\texttt{HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE]},\\ &\texttt{SHORT(?,?)[ACK,MAX_DATA,MAX_STREAM_DATA]},\\ &\texttt{SHORT(?,?)[ACK,STREAM]}, \texttt{SHORT(?,?)[ACK,HANDSHAKE_DONE]}\}\end{aligned}$$

The first four symbols are used to open the connection, perform the handshake, and transmit data, and the last three symbols allow us to model properties related to flow control. We focus on this alphabet, as due to QUIC's numerous packet and frame types, choosing to learn the behavior of, for example, all packet types containing up

to 3 frames, would already give us an alphabet with over 100,000 symbols. Although, in theory, we could learn over such large alphabets, the learning process would take an infeasible amount of time. As we will show in the rest of the section, these seven symbols are enough to capture the main connection establishment, handshake, data transmission, and flow control behavior.

To build the translation pair (α, γ) in the Adapter, we use *QUIC-Tracker* [PDB18] as a reference implementation and instrument it to convert between abstract and concrete symbols. Specifically, we have implemented the format we picked for our abstract alphabet, and whenever our Adapter receives an abstract request from the learner, we use QUIC-Tracker's state to determine whether there is a valid concrete packet matching the abstract packet that should be sent from the current state. If so, we send this queued packet, if not, we request that QUIC-Tracker sends a new packet matching our abstract requests, filling in the concrete values required to make it a valid packet under the current state. QUIC-Tracker then encodes and sends the created packet to the Implementation we are learning a model for and waits for a response. Once this response is detected, it is abstracted and sent back to the learner as a response.

The QUIC Adapter is comprised in total of over 10,000 lines of Go for its reference implementation, however the instrumentation requires only an additional 2,000 lines of code, which we believe is a relatively low cost for PROGNOSIS to handle a protocol as complex as QUIC. Furthermore, the same instrumentation can be used to learn models of all the QUIC implementations. When ran on these three implementations PROGNOSIS could learn models for two of the three implementations (see Section 3.5.6 for explanation). The two models had 12 and 8 states, and 84 and 56 transitions, respectively. Learning took 24,301 queries for one implementation and 12,301 queries for the other. Having these two models enables reasoning with much fewer traces: for the alphabet $\widehat{\Sigma}$ above there are over 300 million traces of length up to 10, however for the same bounded equivalence guarantees [Fuj+91], we only had to check 1210 and 715 of those traces, respectively, for the two implementations. This reduction is gained because the structure of the automata can be used to guide

the equivalence check procedure, rather than using full closed-box exhaustive search.

Table 3.1: Summary of PROGNOSIS results on QUIC implementations.

Target	Model	Bug Type	Detection
QUIC RFC	Learned	Underspecification	Equiv. Check
Cloudflare Quiche	Learned + Synth.	—	—
Google QUIC	Learned + Synth.	Unimpl. Behavior	Equiv. Check + Manual
Facebook mvfst	Partial	Security Vulnerability	Nondet. Check
Quic-Tracker	Learned	Logic Error	Manual Analysis

In Table 3.1 we summarize the result of applying PROGNOSIS to the QUIC implementations mentioned above. In the rest of the section, we describe these issues in more detail, and how they were unveiled using a number of formal and informal analysis of the models generated by PROGNOSIS.

3.5.5 Issue 1: RFC Imprecision

Analysis Technique and Outcome When comparing the models learned for different implementations, we discovered that these models, as reported in Section 3.5.4, had vastly different sizes. We manually explored the models to identify the key differences in structure and found that different implementations were not consistent on what to do if a client resets the *Packet Number Spaces* when retrying a connection. We reported this anomaly to the IETF QUIC Working Group who issued a fix [Tik20] on the next version of the specification to make the expected behavior clearer.

Underlying Issue As QUIC was undergoing standardization, its RFC was a living document still being perfected. One of the guiding principles for this document is the Robustness Principle [Pos80] that was already employed in the development of TCP. The principle states that a system should be strict about what it sends, and liberal on what it receives. As such, the QUIC RFC is perhaps a looser specification than those used for formal verification, and certain aspects of the specification are ambiguous.

Our reported inconsistency unveiled an ambiguity in the behavior of `RETRY` packets. `RETRY` packets are a special kind of packet used by a server to verify the source address of the received packet. It is called `RETRY` as the client should retry to open the connection with the `RETRY` token received from the server. Most

importantly, our report led to a discussion on the topic by the RFC maintainers, and a fix was issued to clarify that a server *MAY* abort the connection when a client resets their Packet Number Spaces [Tik20].

3.5.6 Issue 2: Nondeterminism in Connection Closure

Analysis Technique and Outcome As described in Section 3.4, PROGNOSIS ensures that the answer to every Learner query has a deterministic response via the nondeterminism check. During this check, we found that it was possible to have Facebook's mvfst implementation close the connection and remain in a state where it will not always respond with RESETs to subsequent packets.

Underlying Issue For backwards compatibility reasons, QUIC packets are carried over the UDP protocol. While TCP has been vastly optimized in devices ranging from internet middle-boxes such as switches and routers to endpoint devices such as servers and clients, UDP has not received the same level of attention, mostly due to TCP being the *de facto* protocol for the transport layer.

To understand this bug, we need to introduce a specific QUIC frame: HANDSHAKE_DONE is a signaling frame sent by the server (and only the server) to notify the client that the handshake is now complete, and it can proceed to transmit its data. We include this frame in the input alphabet used by the Learner, and as such the learned model will depict what happens in case the client sends this server-only frame at different states of the system. The ideal response is that, as this frame should never be sent by a client, the server would treat receiving this frame as a protocol violation error, immediately close the connection, and any further packets sent on the same connection *may* be met with a RESET packet, a decision that is up to the developer to make. The RESET packet type is a last case resort that an endpoint can use to notify the other side that this connection no longer exists, even when data transmissions keys are no longer available.

PROGNOSIS detected that if the client starts a connection as usual with INITIAL(?,?)[CRYPTO], and instead of finishing the handshake process for connection establishment, sends a HANDSHAKE_DONE frame to the server with the HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE] symbol, this packet will be met with

a CONNECTION_CLOSE frame, effectively closing the connection. However, after testing the same packet sequence repeatedly, it found that if the client keeps sending packets, it is only in 82% of the responses that packets are met with a RESET.

This behavior discovered in our analysis is *erroneous*: a specific implementation may choose to use RESET packets, but it must be *consistent* in the decision. It cannot nondeterministically switch between sending and not sending packets. Furthermore, this RESET behavior has no back-off mechanism, meaning that a client can exploit this bug to request new packets from the server on demand. The client could keep sending the exact same packet, with no computation needed, and the server would have to produce new RESET packets every time it receives this unexpected packet. This behavior, coupled with the fact that RESET packets are relatively small, means that system-level UDP optimizations are not triggered, resulting in sending each RESET packet being an expensive operation. If exploited, this behavior can be used in a *Denial-of-Service* attack against the server. This critical bug has been acknowledged by the developers and fixed.

3.5.7 Issue 3: Inconsistent Port on RETRY in QUIC-Tracker

Analysis Technique and Outcome After learning the model of one of the implementations we detected a discrepancy (in terms of traces) in how two different implementations handle the *Retry Mechanism*: if the server sent a RETRY packet the model would transition to a state where connection establishment was impossible. This greatly contrasted with behavior learned in another model.

One of the properties of a confirmed erroneous trace is that either the Implementation or the Adapter are behaving unexpectedly. In this case, it was the reference implementation, QUIC-Tracker, that had a bug in its retry mechanism, and only one of the target implementations considered it critical enough to not allow connection establishment to take place.

Underlying Issue QUIC's Retry Mechanism, as introduced in Section 3.5.5, is a way for the server to validate that a packet is indeed being sent from a specific IP address and port, and not just an attacker replaying packets from a spoofed source. The client starts by sending a ClientHello as usual, but this time the server replies

to it with a RETRY packet. This packet has a unique token which the client must take and send in a new ClientHello from the same IP address and port. The server then checks if the Retry Token matches the one previously sent to confirm these packets are being sent from the claimed source. Our reference implementation was correctly returning the token; however it was doing so with a new UDP socket using a random free port instead of the one we were using before. As such, the token would be sent from a different port, and address validation would fail, interrupting the handshake.

3.5.8 Issue 4: Stream Data Blocked Bug in Google QUIC

Analysis Technique and Outcome Using the extended models of Section 3.3.2, we synthesized an extended Mealy machine (shown in Appendix A.2.1) that describes how the Maximum Stream Data field in the STREAM_DATA_BLOCKED frame changes over the states. By inspecting this model we were able to detect that this field always has the value 0, and is never updated, even when the stream gets blocked.

Underlying Issue One of the key features of QUIC is flow control. For this, QUIC makes use of a series of frames dedicated to coordinating flow limits between the two endpoints. One such frame, the STREAM_DATA_BLOCKED frame, is responsible for alerting the other endpoint when we would like to send data on a stream, but the flow control limits imposed on us by the endpoint itself do not allow us to do so. This frame has two fields, a Stream ID field indicating which stream is blocked, and a Maximum Stream Data field, indicating at what offset of the data the block happened. We found that although one of the implementations used these frames and did not transmit data over the agreed limits, it did not set the Maximum Stream Data correctly, and instead used the constant 0. The developers have confirmed that this section of the specification was incorrectly implemented. They explained that 0 was a placeholder set during initial development, which was not updated.

Evaluation The case studies provide a positive answer to RQ1. PROGNOSIS was successfully applied across different protocol implementations while retaining the same overall learning and analysis architecture: the TCP case study demonstrates that the framework can reproduce previously obtained modeling detail with limited adaptation, while the QUIC case studies reuse a common protocol-specific Adapter

across several independent implementations. More importantly, the models produced exposed several forms of unintended behavior, including an ambiguity in the QUIC specification, nondeterministic behavior leading to a bug in mvfst, an implementation error in QUIC-Tracker, and an incorrect flow-control field in Google QUIC. Together, these results show that PROGNOSIS provides a reusable closed-box approach for learning models of protocol implementations that are sufficiently informative to support the discovery and diagnosis of defects and specification ambiguities.

3.6 Related Work

As mentioned in Section 1.3, automata learning has been applied to a range of communication and security protocols [ASV10; ARP13], including network protocols [De 16; FJV16; Arg+16; Fit+17; Fit+20]. PROGNOSIS improves on previous work in several directions: first, the architecture of PROGNOSIS is parametric with respect to reference implementation, an aspect that greatly decreases the amount of work and expertise needed to analyze one protocol implementation; second, PROGNOSIS allows a user to swap different implementations of the same protocol seamlessly by requiring only a socket change; third, PROGNOSIS uses synthesis to enrich the models with data and enables a more fine-grained control in the type of properties and details one can analyze. PROGNOSIS is the first automata learning tool used to analyze several QUIC implementations.

Other approaches to analyze QUIC include work by McMillan and Zuck [MZ19] who manually build a formal specification of the wire protocol in the Ivy language, and then use it for test generation and find a range of bugs in different implementations. In contrast, the approach used in PROGNOSIS automates the building of a finite state model guided by an abstract alphabet. The models learned by PROGNOSIS can also be used for test generation, and crucially one does not need to manually encode the protocol logic, including complex cryptographic components, to use PROGNOSIS. McMillan and Zuck [MZ19] specify the full protocol state including the security handshake, with unbounded data. The models we learned with PROGNOSIS abstract away details not covered by the alphabet or the registers we chose to synthesize.

Closely related to the work of McMillan and Zuck [MZ19], but only for TCP, is the work of Bishop et al. [Bis+19] using symbolic model checking.

Other approaches to analyze protocols include building a *correct-by-construction* reference implementation [Bha+17; Del+21] and then using it when testing a new implementation. This approach has the advantage of having every component in the reference implementation formally verified. However, the amount of manual work to build such implementation is out of reach of most development teams and, moreover, it will be making specific choices in terms of the specification.

PROGNOSIS is closed-box: it does not assume access to the code of the implementation being analyzed. Open-box approaches to analyze QUIC also appeared in the literature [RSW18; GPP20]. These works focus on testing protocol compliance using symbolic execution. However, subtle bugs related to ambiguities in the RFC or differences between implementations (like the ones detected by PROGNOSIS and described in Section 3.5) would not be detected easily.

Finally, PROGNOSIS complements differential testing [McK98]: the learned models as well as the Adapter can be used to create high-quality test cases that trigger complex behaviors of the protocols, something that is typically hard in a closed-box setting. Furthermore, PROGNOSIS produces models that can be inspected by protocol designers, something that differential testing does not directly help with.

In summary, our work focuses on developing a modular, reusable, and flexible framework, where different formal analysis techniques can easily be made accessible for different protocols and implementations thereof.

3.7 Discussion

PROGNOSIS has been successfully used to analyze TCP and QUIC implementations and has found several bugs in different mainstream QUIC implementations. One of the key contributions of PROGNOSIS is the use of a reference implementation to remove the burden of having to implement the protocol logic from the user. This step was required in previous approaches based on automata learning. Though this contribution means that less experienced users have easier access to PROGNOSIS, it

still requires some knowledge of instrumenting a reference implementation.

An interesting direction for future work is investigating whether one can more easily develop an Adapter through custom trained neural networks, in a neurosymbolic fashion [Bar+25]. Developing the abstraction function α is relatively easy: we merely need to remove enough detail that we do not want the learner to concern itself with. Producing the concretization function γ is indeed the hard part. However, perhaps one can use sufficient example mappings of α on fuzzed network traces to *train* a neural network on producing a suitable inverse function γ.

Over the development of QUIC, PROGNOSIS has identified bugs that have subsequently been corrected. However, verifying this requires learning new models, to check whether indeed the behavior of the protocol has been changed. Doing so from scratch proves rather wasteful: it is unlikely the vast majority of the protocol has changed at all, and thus it would make more sense to be able to *reuse* previous models, and learn *incrementally*. Existing works on *adaptive* automata learning aim to do precisely this [DMD19]. However, such algorithms tend to be based on much simpler ones, such as the $L^\star$ algorithm [Ang87], which by now has been surpassed by vastly more efficient ones such as KV [KV94], TTT [IHS14], and $L^\sharp$ [Vaa+22]. Chapter 4 focuses precisely on the goal of contributing a more efficient (classification tree-based) algorithm for adaptive learning.

Finally, the models PROGNOSIS can currently synthesize do not encompass any form of *environment quantities*, which are interesting in a networking context to capture, for example, congestion, latency, or memory usage properties. Extending PROGNOSIS to richer quantitative models is perhaps the most challenging yet impactful direction for future work, as it will require significant advances on the design of learning algorithms. Recent developments in active learning of weighted automata [Van+20; BM15; DJ25; Ari+25] provide a good starting point for this direction, however they suffer from limitations such as non-minimality, and potential non-termination. In Chapter 6 we consider a new weighted automata learning algorithm that aims to make progress in this front.

Chapter 4

Tree-Based Incremental Automata Learning

Classic active model learning algorithms like $L^\star$ [Ang87], KV [KV94], and TTT [IHS14] exist for a number of automata types (deterministic, input-output, weighted, register) and have enabled analysis of numerous systems by providing faithful models to be used in model checking. However, these algorithms suffer from a common issue: Systems often change faster than we can learn their models, as there is an inherent assumption that the learning process is running from scratch every time. As such, keeping a model up-to-date becomes quite challenging, often needing manual intervention from an expert. This does, however, introduce the chance of producing a model that is not actually correct, while being an extremely laborious task for any moderately sized complex system.

In this chapter, we present IKV, an *incremental* automata learning algorithm that can cope with evolving systems more effectively, and does not have to restart the learning process when a change occurs, providing a gain in efficiency when learning systems that undergo changes at known moments in time.

The work in the present chapter can be seen as advancing the state of the art in *adaptive* automata learning [GPY02; Cha+08; Win+13; HMV18; DMD19]. Whereas previous work adapted the $L^\star$ algorithm to reuse part of a data structure from a previous run, our adaptive algorithm is the first to use the more efficient *classification tree* data structure, central to the Kearns–Vazirani (KV) algorithm [KV94]. Although

we focus on developing an algorithm for deterministic finite automata (DFAs), the techniques we developed can be transferred to other automata models. We expect this work and subsequent developments on adaptive learning will bring us closer to meeting the needs of employing lightweight formal methods in fast-moving industries, and in large evolving systems such as the protocol implementations learned in Chapter 3.

The remainder of this chapter is organized as follows. In Section 4.1 we cover preliminary material on classification trees and their use for automata learning. In Section 4.2 we introduce and argue correctness of our incremental learning algorithm for DFAs, which targets evolving systems whose mutation points are known during learning. In Section 4.3 we benchmark and evaluate the efficiency of the algorithm, and we demonstrate its effectiveness in learning evolving systems. We conclude in Section 4.5 with a discussion of further directions on adaptive automata learning.

4.1 Learning with a Classification Tree

When designing a learning algorithm, one of the key aspects to consider is how we store the information we acquire over time. Learning then becomes a matter of being able to extend this structure with as little queries as possible, and transforming the data into a hypothesis automaton.

Angluin's $L^\star$ algorithm uses an observation table to record the set of discovered state access sequences S, together with the set of suffixes E that prove differences in behavior between the access sequences. This is achieved by having S as the row labels of a table, E as its column labels, and each cell with row label $s \in S$ and column label $e \in E$ filled with $mq(s \cdot e) \in \mathbb{2}$. The distinct rows of this table represent then the Myhill–Nerode equivalence classes of the target language, i.e., the distinct states of the target DFA. By extending the row labels with each letter of the alphabet $(S \cdot \Sigma)$, transitions of a hypothesis DFA can be determined, and the initial and final states follow using the empty word row and column, respectively. For a deeper presentation of $L^\star$'s observation table, see Section 2.3.

Although the table stores all information needed to form a DFA, it may contain

data that is duplicated, and even unnecessary. This is because each suffix $e \in E$ is considered in classifying each access sequence $s \in S$, which guarantees that all states are distinguished. However, we only need to be able to distinguish between every pair of states, and, via the Myhill–Nerode congruence, this takes at most 1 suffix per pair. As such, and as done by Kearns–Vazirani in their learning algorithm [KV94], a binary classification tree is a more appropriate data structure, where each leaf in the tree is labeled by a word $s \in S$, and the nodes labeled with words $e \in E$ provide for each pair of leaves a suffix to distinguish them.

The learning algorithm we introduce later, similarly to the classic KV algorithm, uses *classification trees* as its base data structure. Not only is minimizing the data needed to build a DFA important to minimize the number of queries used in learning, in our case we are also minimizing the data that we would need to recheck and correct as our target mutates.

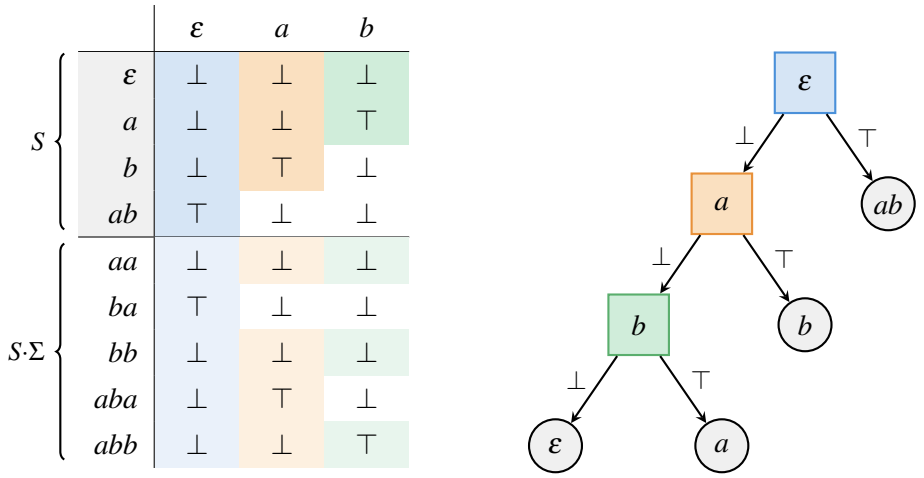
Definition 4.1.1 (Classification Tree [KV94])**.** *A* classification tree *is a binary tree with labels in* $\Sigma^\star$*. Formally, classification trees are defined by the grammar:*

$$\mathsf{CT} \longrightarrow \mathsf{Node}\ \Sigma^\star\ \mathsf{CT}\ \mathsf{CT} \mid \mathsf{Leaf}\ \Sigma^\star$$

△

Here, a node contains a classifier $e \in \Sigma^\star$, and $\bot$-child and $\top$-child subtrees, and a leaf contains a single access sequence $s \in \Sigma^\star$. The child subtrees are named this way because of how the classifier e distinguishes access sequences $s_\bot$ and $s_\top$ present in the respective subtrees: $mq(s_\bot \cdot e) = \bot$ and $mq(s_\top \cdot e) = \top$. In particular, this holds for every pair of leaves and their *lowest common ancestor* node, the root node of the smallest subtree containing both leaves.

Example 4.1.2. *Consider the target language* $\mathcal{L} \triangleq \{w \in \{a,b\}^\star \mid |w|_a$ *is odd* $\wedge$ $|w|_b$ *is odd*$\}$*, which has 4 Myhill–Nerode equivalence classes, and therefore can be computed by a 4 state DFA. Below we show final data structures from learning this language, using* $\mathrm{L}^\star$ *(a, left) and* KV *(b, right).*



		ε	a	b
S	ε	$\bot$	$\bot$	$\bot$
	a	$\bot$	$\bot$	$\top$
	b	$\bot$	$\top$	$\bot$
	ab	$\top$	$\bot$	$\bot$
$S\cdot\Sigma$	aa	$\bot$	$\bot$	$\bot$
	ba	$\top$	$\bot$	$\bot$
	bb	$\bot$	$\bot$	$\bot$
	aba	$\bot$	$\top$	$\bot$
	abb	$\bot$	$\bot$	$\top$

(a) $L^\star$ observation table

(b) KV classification tree

/ classify/sift via ε / via a / via b unshaded: not queried by KV

We have shaded in the (S portion of) the table, the data points that are equivalently known in the classification tree. Notice that these are sufficient: Were we to remove the information from unshaded cells, the rows would still all be distinguished. The bottom $S\cdot\Sigma$ portion of the table is absent in the tree, as in KV *it is computed lazily when needed. Nonetheless we shade the datapoints that would be queried in this sifting process. While the observation table has $9\times 3 = 27$ cells that need to be filled, only* 21 *of these are equivalently needed by the classification tree.*

Subroutine 4.1: `sift` returns the leaf in a classification tree equivalent to a provided word w.r.t. the equivalence induced by the tree.

Data: Classification tree $tree \in \mathsf{CT}$, membership oracle $mq\colon \Sigma^\star \to 2$, word $w \in \Sigma^\star$.
Result: Equivalent leaf in $tree$.

```
if tree = Node e left right then
    return mq(w·e) ? sift(right, mq, w) : sift(left, mq, w);
return tree;
```

The classification tree is then able to classify every word $w \in \Sigma^\star$ into a specific leaf of the tree, depending on how the target accepts or rejects w concatenated with specific classifiers e in the tree. This is done through *sifting* (Subroutine 4.1), where, starting from the root of the tree with classifier e, we pose the query $mq(w\cdot e)$ and, depending on the result, proceed to the $\bot$-child or the $\top$-child of the node, from

which we continue sifting, until we reach a leaf. The access sequence s of that leaf will then be deemed equivalent to w according to the Myhill–Nerode equivalence classes captured by the tree.

Example 4.1.3. *Consider sifting some of the words in the $S \cdot \Sigma$ portion of the observation table in Example 4.1.2. For ba, the root classifier ε asks $mq(ba \cdot \varepsilon) = \top$, and hence immediately reaches the leaf ab. Thus:*

$$\texttt{sift}(tree, mq, ba) = \mathsf{Leaf}\ ab$$

For aba, we first have $mq(aba \cdot \varepsilon) = \bot$, taking the $\bot$-branch of the root, followed by $mq(aba \cdot a) = \top$, which reaches the leaf b. Sifting abb instead follows the deepest path in the tree:

$$mq(abb \cdot \varepsilon) = \bot \qquad mq(abb \cdot a) = \bot \qquad mq(abb \cdot b) = \top$$

and therefore reaches the leaf a. Finally, aa follows the same path, with the last query returning $\bot$, reaching the leaf ε:

$$mq(aa \cdot \varepsilon) = \bot \qquad mq(aa \cdot a) = \bot \qquad mq(aa \cdot b) = \bot$$

Hence, the tree classifies:

$$ba \sim ab \qquad abb \sim a \qquad aba \sim b \qquad aa \sim \varepsilon$$

Since this is the final classification tree for L, these correspond to the four Myhill–Nerode equivalence classes of the target language. Notice also that sifting only queries the classifiers encountered along a word's path: classifying ba requires only one membership query, while aba requires two and abb and aa require three.

4.1.1 Constructing Hypothesis Automata

The extraction of a hypothesis from a corresponding classification tree, as detailed in Subroutine 4.4, is then semantically the same as in $L^\star$. The set of states in the

automaton are the leaves in the classification tree, which match with the rows of the access sequences of the observation table. These act as representatives of the Myhill–Nerode equivalence classes found thus far, as separated by the distinguishing suffixes stored in the tree's inner nodes, or the columns of the observation table. The accepting states are the access strings that on the empty suffix ε are accepted, which in the tree are the leaves of the $\top$-subtree of the ε node, and in the table are the rows that have a $\top$ value for the ε column. The initial state is similarly, the class of the ε string, i.e., its equivalent leaf in the tree, or row in the table.

Subroutine 4.2: `leaves` returns the set of leaves in a given tree.

Data: Classification tree *tree* $\in \mathsf{CT}$.

Result: Set of leaves in *tree*.

```
1 if tree = Node e left right then
2 |   return leaves(left) ∪ leaves(right);
3 return {tree};
```

Subroutine 4.3: `child` returns the ($\bot$/$\top$)-child of a provided tree node.

Data: Tree node $p \in \mathsf{Node}\ \Sigma^\star\ \mathsf{CT}\ \mathsf{CT}$, $b \in \mathbb{2}$

Result: b-child of p.

```
1 if p = Node e left right then
2 |   return b ? right : left;
```

Finally, the logic behind computing transitions also follows the canonical DFA (Definition 2.2.8) based on successor equivalence classes. While L*'s observation table eagerly computes the equivalence class of every successor state, KV performs this on-the-fly by sifting each extended string. I.e., to determine the state / equivalence class that we reach from a state represented by $s \in S$, on symbol $a \in \Sigma$, we sift the string $s \cdot a$ to determine which state it is equivalent to in the tree.

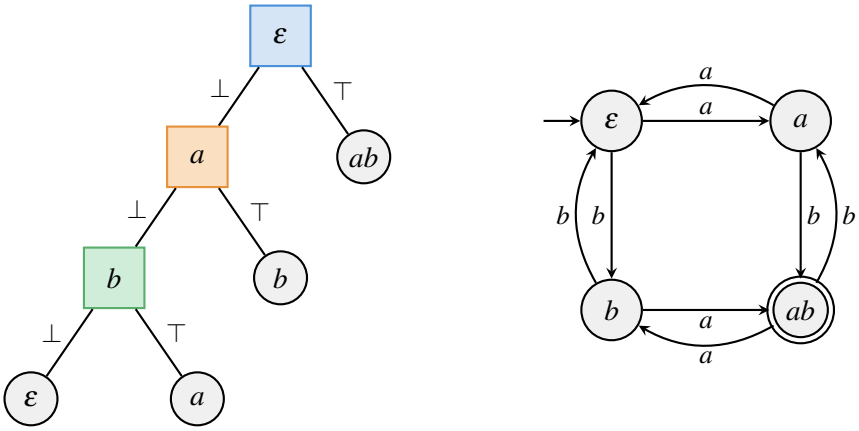
Subroutine 4.4: `buildHyp` extracts a DFA from a classification tree.

Data: Classification tree $tree \in \mathsf{CT}$, membership oracle $mq\colon \Sigma^\star \to \mathbb{2}$.

Result: Hypothesis DFA $\mathcal{H}$ w.r.t the tree $tree$.

```
1 if tree = Leaf l then
2 |   return mq(l) ? A_Σ^⊤ : A_Σ^⊥;
3 q_0 ← sift(tree, mq, ε);
4 Q ← leaves(tree);
5 F ← {Leaf l ∈ leaves(tree) | mq(l) = ⊤};
6 for Leaf l ∈ Q, a ∈ Σ do
7 |   δ(Leaf l, a) ← sift(tree, mq, l·a);
8 return ⟨Q, q_0, Σ, δ, F⟩;
```

Example 4.1.4. *Below we consider the classification tree learned from Example 4.1.2 (a, left) and its corresponding automaton (b, right) constructed from* `buildHyp`.



(a) Classification tree (b) Hypothesis automaton

Note how expectedly, the leaves of the tree form the states of the automaton, and sifting of one-letter extensions of said leaves gives is the transitions in the automaton. The ε leaf formed the initial state, while the root $\top$-child leaves are the accepting states, in this case only state ab, and the $\bot$-child leaves are the rejecting states.

4.1.2 Refining From Counterexamples

With a hypothesis extracted from the classification tree, we can now pass this to an equivalence oracle to determine whether the hypothesis is correct. If not, we will receive a counterexample word that we can use to improve the classification tree.

The algorithm does this through the logic that, given that every hypothesis classifies the empty string ε correctly, and by definition the current hypothesis classifies the counterexample c incorrectly, there must be a prefix of c for which the classification first diverges. In terms of states, this then means that we are taking a transition into a state that is accepting in the hypothesis, and rejecting in the target, or vice versa. This is fixed by realizing that the state we take this transition from then must actually be two different states. The algorithm then uses this logic in `updateTree` (Subroutine 4.7) to split the state into two at the tree level, turning a leaf into a node with two leaves, one representing the new state discovered by the counterexample.

Subroutine 4.5: `LCA` returns the classifier of the lowest common ancestor node of two provided leaves in a classification tree.

Data: Classification tree $tree \in \mathsf{Node}\ \Sigma^\star\ \mathsf{CT}\ \mathsf{CT}$, first leaf l_a, second leaf l_b.
Result: LCA node clasifier e for l_a, l_b in the classification tree $tree$.

```
1 if tree = Node e left right then
2 |   if l_a, l_b ∈ leaves(left) then
3 |   |   return LCA(left, l_a, l_b);
4 |   if l_a, l_b ∈ leaves(right) then
5 |   |   return LCA(right, l_a, l_b);
6 |   return e;
```

Subroutine 4.6: `split` splits a leaf in the tree into a node with 2 child leaves, one of them a new leaf introduced to the tree.

Data: Classification tree $tree \in \mathsf{CT}$, current leaf l being split, label $w \in \Sigma^\star$ for the new leaf, classifier $e \in \Sigma^\star$ for the new node, $b \in \mathbb{2}$ indicating whether the new leaf is a $\top$-child.
Result: Updated classification tree $tree$.

```
1 if tree = Leaf w' then
2 |   return b ? (Node e w' w) : (Node e w w');
3 if tree = Node e' left right then
4 |   if l ∈ leaves(left) then
5 |   |   return Node e' split(left, l, w, e, b) right;
6 |   else
7 |   |   return Node e' left split(right, l, w, e, b);
```

Intuitively, given a current classification tree $tree \in \mathsf{CT}$, its equivalent hypothesis $\mathcal{H}$, and a counterexample c, for each prefix $c[0..i]$ of c we compute leaves s_i and $\hat{s}_i$. These are the leaves reached by *sifting* the prefix, versus traversing the hypothesis

automaton, respectively. Without loss of generality, let i be the lowest such that $s_i \neq \hat{s}_i$. Then i is the first index for which the prefix of the counterexample, according to sifting via the membership oracle is equivalent to leaf s_i, but according to the transition structure computed from the tree, is equivalent to leaf $\hat{s}_i$.

Subroutine 4.7: `updateTree` with a provided counterexample.

Data: Classification tree $tree \in \mathsf{CT}$, membership oracle $mq\colon \Sigma^\star \to \mathbb{2}$, counterexample $c \in \Sigma^*$.

Result: Updated $tree \in \mathsf{CT}$ refined with one more leaf.

```
1  if tree = Leaf l then
2      return split(tree, tree, c, mq(c));
3  H ← buildHyp(tree, mq);
4  for i ∈ {0, ..., |c| − 1} do
5      s_i ← sift(tree, mq, c[0..i]);
6      ŝ_i ← reach_H(c[0..i]);          /* state reached in H by c[0..i] */
7      if s_i ≠ ŝ_i then
8          e ← c_i · LCA(tree, s_i, ŝ_i);
9          tree ← split(tree, s_{i−1}, c[0..i − 1], e, mq(c[0..i − 1] · e));
10         return tree;
```

How can this be? The hypothesis automaton is only correct w.r.t the assumption that its classification tree represents all Myhill–Nerode equivalence classes. In which case, a leaf's access string indeed fully represents the strings reaching the same state, according to the classifier nodes of the tree. It is then reasonable to compute transitions from a state using a single access string representative of said state. In a classification tree not yet fully grown this assumption breaks down, and the counterexample prefix is simply a witness of a string that actually reaches a different state than computed by the hypothesis. I.e., according to the classification tree's own approximation of the Myhill–Nerode classes, it belongs to a different state than computed by the transition structure.

Then i is the first index for which the prefix of the counterexample reaches provably an incorrect state in the hypothesis. This means we are starting from the same "synchronised" start state, taking the same symbol transitions but reaching

different states. As this is impossible in a *deterministic* finite automaton, it must mean we are actually taking transitions from two different states that in our hypothesis are incorrectly represented as one.

To correct this, we split the leaf of the last correct state (s_{i-1}) by turning it into a node with two child leaves. One of these will be the original s_{i-1} leaf; the other will be the new discovered state with access sequence $c[0..i-1]$. The classifier of the new node is set to $c_i \cdot d$, where c_i is the i-th symbol of c, and d is the suffix distinguishing s_i and $\hat{s}_i$ in the tree.

The reader may have noticed that this is precisely equivalent to the process of detecting, and correcting inconsistencies of an $L^\star$ observation table (see Definition 2.3.5), just performed from the perspective of the classification tree.

4.1.3 The Kearns–Vazirani Algorithm

We now have all the components needed to provide the classic KV algorithm in Algorithm 4.8. This algorithm uses the `buildHyp` routine explained above to build a DFA from a classification tree, as well as `updateTree` (Subroutine 4.7) to extend the classification tree on receipt of a counterexample from an equivalence oracle.

Algorithm 4.8: Kearns–Vazirani Algorithm [KV94, Sec. 8.3.5]

Data: Alphabet Σ, membership oracle $mq\colon \Sigma^\star \to \mathbb{2}$ and equivalence oracle $eq\colon \mathsf{DFA} \to \mathbb{1} + \Sigma^\star$ for language $\mathcal{L}$.
Result: The minimal DFA $\mathcal{H}$ accepting $\mathcal{L}$.

```
1 tree ← Leaf ε;
2 H ← buildhyp(tree, mq);
3 cex ← eq(H);
4 while cex ∈ Σ* do
5 |   tree ← updateTree(tree, mq, cex);
6 |   H ← buildHyp(tree, mq);
7 |   cex ← eq(H);
8 return H;
```

Correctness and termination guarantees of the algorithm are then very similar to $L^\star$'s. The learn loop of the KV algorithm is essentially the same, just with operations on classification trees rather than observation tables: The learner produces a hypothesis automaton from its data structure, and poses an equivalence query to

the teacher. If one is returned, the data structure is refined such that it discovers (at least) one new Myhill–Nerode equivalence class by refining an existing one into two, and the loop repeats until there are no counterexamples from the teacher.

Theorem 4.1.5 (KV Correctness & Termination [KV94, Section 8.3.6])**.** *Given a minimally adequate teacher $\langle mq_{\mathcal{L}}, eq_{\mathcal{L}} \rangle$, the* KV *algorithm learns the canonical DFA $\mathcal{A}_{\mathcal{L}}$ in finite time, with at most $|\mathcal{A}_{\mathcal{L}}|$ equivalence queries, and membership queries polynomial-bound on $|\mathcal{A}_{\mathcal{L}}|$ and the length of the longest counterexample.*

An interesting difference to L*, is that while the counterexample handling of L* guarantees that the counterexample itself is fixed, through finding however many new states are needed to do so, the KV algorithm guarantees only that one new state is found towards that goal. As such, many implementations of KV reuse the same counterexample provided until it is fixed, i.e., until all the needed states to fix the counterexample are found. Note however that this is an optimization, and not needed for termination: Finding one new Myhill–Nerode equivalence class per counterexample still guarantees a finite number of iterations of the learning loop.

4.2 Learning Evolving Systems Incrementally

We now present the IKV algorithm for learning updates to a previous model without discarding all behavior learned so far. It is also able to detect and remove behavior that no longer holds without input from an expert in the system.

Classic automata learning algorithms "learn", in the sense of reconciling a model of the system in the face of new knowledge, in a purely *additive* way. A hypothesis is constructed such that it is consistent with the finite information acquired so far, extrapolating minimally to provide a complete automaton. When a counterexample is provided against this hypothesis, it does not contradict the information seen so far, rather it proves that the model must go further, and be slightly more complex to extrapolate into the correct automaton. However, when learning a system that may have changed in unknown ways, this is no longer the case. A counterexample indeed may directly conflict with information seen previously, which requires a level of reconciliation that classic learning algorithms are not capable of.

Adaptive learning algorithms are able to do this by, at specific points in the learning process, detecting outdated knowledge that is no longer true of the system. Thus, they can deal with languages that mutate over time, adapting to changes by either trimming outdated behavior or distinguishing new behavior.

Our adaptive learning algorithm is targeted at systems with discrete changes, such as version-controlled systems. Specifically, the system evolves at discrete known points, forming a stream of target systems. Our *incremental learning algorithm* for DFAs, presented in Algorithm 4.11, is based on KV. As a first crucial difference, the incremental algorithm uses a previous learned model as its starting point. As such, it cannot just acquire new information; it also needs to be able to trim outdated behavior. This is done by `minimizeTree` (Subroutine 4.10), which prunes an initial tree by removing all leaves that are no longer represented by their reported access sequence. It achieves this by sifting every access sequence down the tree, removing leaves whose access sequences do not sift back into themselves.

Subroutine 4.9: `removeLeaf` removes a given leaf from the tree.

Data: Classification tree *tree*, leaf $l \in \texttt{leaves}(tree)$ to be removed.
Result: A valid classification tree *tree* with l removed.

```
1 if tree = Node e left right then
2     if left = l then
3         return right;
4     if right = l then
5         return left;
6     return Node e removeLeaf(left, l) removeLeaf(right, l);
```

Subroutine 4.10: `minimizeTree` trims the tree from redundant leaves.

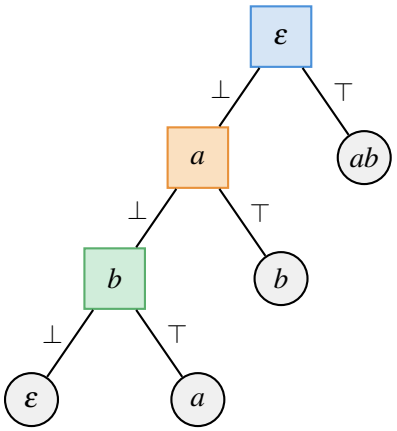
Data: Classification tree $tree \in \mathsf{CT}$, membership oracle $mq\colon \Sigma^\star \to \mathbb{2}$.
Result: A minimized classification tree.

```
1 for Leaf l ∈ leaves(tree) do
2     s ← sift(tree, mq, l);
3     if s ≠ Leaf l then
4         tree ← removeLeaf(tree, Leaf l);
5 return tree;
```

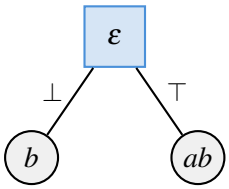
This guarantees not only that the leaves left in our tree form a valid approximation of the new equivalence classes of the mutated language, but also that every

leaf in the tree is unique w.r.t. this new Myhill–Nerode congruence. If a pair of leaves were equivalent, then both their access sequences would sift into only one leaf, leaving the other leaf to not sift back into itself, and causing it to be removed.

Example 4.2.1. *Suppose now the target language evolves from* $\mathcal{L} \triangleq \{w \in \{a,b\}^\star \mid |w|_a \text{ is odd} \wedge |w|_b \text{ is odd}\}$ *to* $\mathcal{L}' \triangleq \{w \in \{a,b\}^\star \mid |w|_a \text{ is odd}\}$*, dropping the parity constraint on b. Under* $\mathcal{L}'$*, the leaves* ε *and b become equivalent (both have even* $|w|_a$*), as do a and ab (both have odd* $|w|_a$*).*



(a) Classification tree for $\mathcal{L}$ ($|w|_a$ *is odd* $\wedge$ $|w|_b$ *is odd*)



(b) Pruned classification tree for $\mathcal{L}'$ ($|w|_a$ *is odd*)

The former leaf ε *now sifts into the leaf of b, and a into ab. As such,* `minimizeTree` *prunes these leaves, retaining only the Myhill–Nerode equivalence classes that are still present in the new language.*

On top of this change to the main learning loop (Algorithm 4.11), we also adapt the supporting subroutines, such as `buildHyp` and `updateTree`, so that they remain correct not only for the classic KV algorithm, where classification trees only grow, but also for IKV, where trees may shrink. First, all subroutines now support single-leaf classification trees as a base case. The classic KV versions assume a tree with at least two leaves, since the tree only starts being used after the trivial single-state hypothesis has been refuted, and thereafter only grows.

Second, the hypothesis construction procedure `buildHyp` can no longer rely on a hard-coded ε leaf to identify the initial state. In the incremental setting, `minimizeTree` may remove that leaf if the empty word now sifts to another surviving representative. The more robust construction is therefore to determine the

Algorithm 4.11: IKV Algorithm

Data: Fixed alphabet Σ, optional previous classification tree $tree \in \mathbb{1} + \mathsf{CT}$, membership oracle $mq\colon \Sigma^\star \to \mathbb{2}$ and equivalence oracle $eq\colon \mathsf{DFA} \to \mathbb{1} + \Sigma^\star$ w.r.t the language $\mathcal{L}$.
Result: The learned minimal DFA $\mathcal{H}$ for the language $\mathcal{L}$.

```
1  if tree ∈ CT then
2  |  tree ← minimizeTree(tree, mq);
3  else
4  |  tree ← Leaf ε;
5  H ← buildhyp(tree, mq);
6  cex ← eq(H);
7  while cex ∈ Σ* do
8  |  tree ← updateTree(tree, mq, cex);
9  |  H ← buildHyp(tree, mq);
10 |  cex ← eq(H);
11 return H;
```

initial state by sifting ε through the current tree, i.e. by computing the leaf currently equivalent to the empty word.

Similarly, `minimizeTree` may remove the root classifier node labeled ε as a consequence of pruning a child and promoting its sibling subtree. Since the classic KV construction uses this root classifier to determine which hypothesis states are accepting, IKV instead determines this directly: A leaf is accepting precisely when a membership query on its access sequence returns $\top$.

4.2.1 Correctness and Termination

As the algorithm only terminates with a hypothesis that is correct according to an equivalence query, correctness follows from termination.

Termination of the original KV algorithm relies on the following key property of classification trees: for every subtree of the form $\mathsf{Node}; e; \mathit{left}; \mathit{right}$, each leaf $s \in \Sigma^\star$ of *left* satisfies $s \cdot e \notin \mathcal{L}$ and every leaf $s \in \Sigma^\star$ of *right* satisfies $s \cdot e \in \mathcal{L}$. In other words, each internal classifier correctly separates the leaves in its left and right subtrees with respect to the current target language.

After `minimizeTree`, this property holds relative to the new target language, as any leaf whose access sequence no longer follows the path identifying that leaf is removed. Thus, when the main loop starts, the remaining leaves are valid

representatives under the current membership oracle, and the tree can again be used as a classification tree for the current target. Together with the generalized `buildHyp` described above, this gives the invariants needed to resume the standard KV refinement argument.

When a counterexample *cex* is found, refinement proceeds as in the original KV algorithm. The counterexample identifies a transition whose source leaf must in fact represent at least two distinct equivalence classes; `updateTree` then splits that leaf using an appropriate classifier and thereby adds one new leaf while preserving the classification-tree invariant above. Since the leaves represent distinct Myhill–Nerode equivalence classes of $\mathcal{L}$, and each counterexample refinement strictly increases the number of leaves, this step can occur only finitely many times. The number of leaves is bounded by the number of equivalence classes of the target language, i.e. by the number of states of its minimal DFA.

4.3 Experimental Evaluation

We now aim to evaluate the efficiency of our algorithm, IKV, in the setting of learning evolving systems incrementally. In this section we describe our implementation, and develop research questions to compare its performance to both its classic counterpart KV and the state of the art in adaptive automata learning at the time of development.

4.3.1 Implementation

As is the case for many automata learning algorithms, we have implemented the IKV algorithm as an extension of LearnLib [FHS25], a Java library providing a rich support ecosystem for automata learning, from the algorithms themselves, to oracle implementations, caching layers, different counterexample processing algorithms, and more. Adding the IKV algorithm to LearnLib required approximately 700 lines of Java code, making use of this existing infrastructure.

Another advantage of using LearnLib, is that it allows us to make a fair comparison with implementations of other algorithms. For our baseline comparison, the KV algorithm is already implemented in LearnLib, making use of shared infrastructure with IKV. For our comparison with the state of the art, we identified the PDL* algo-

rithm [DMD19] as the latest and most performant incremental algorithm, at the time of the development of IKV. The PDL* algorithm has an original implementation in (an unofficial fork of) LearnLib. We incorporated this code into the same version of LearnLib as IKV, to ensure they share the same common infrastructure.

4.3.2 Research Questions and Methodology

We are interested in answering the following research questions (RQs) on the efficiency of our algorithm:

RQ1 How does the IKV algorithm perform compared to the KV algorithm?

RQ2 How does the IKV algorithm perform compared to the PDL* algorithm?

To answer these research questions we have devised a large set of experiments. At a high level, our methodology is centered on considering a set of pairs (t_0, t_{10K}) of target DFAs, which model the evolution of target t_0 into t_{10K}, at a fixed point "in time". We can then study how each algorithm behaves in learning t_0, and at the point of evolution, recovering from the change and producing a correct model for t_{10K}. We now make the details of these experiments more formal, and justify design decisions.

4.3.2.1 Benchmarks

Ideally, we would run our experiments over a standard set of benchmarks [Nei+19], however these currently only cover single target automata, and so are not fit for adaptive learners designed to learn automata that are related in some way, usually due to small evolutions in their behavior. Instead, we designed two families of target automata to benchmark the algorithms. For both of these, we consider t_0 being randomly generated, *canonical* (and therefore minimal) automata of increasing state sizes ($|Q|$), from 10 states to 80 states. We opted for a simple alphabet of 3 symbols ($|\Sigma| = 3$). This is to make the set of experiments manageable, and because the complexity of all considered algorithms is dominated more by the state space, than the size of the alphabet. For the t_{10K} target, we construct them relative to the respective t_0, in the following ways.

For the first family of experiments, which we call the *Mutation Benchmark*, we apply a series of random mutations to t_0 in order to create t_{10K}. Specifically, we

randomly divert a transition, add a state, remove a state, and flip the acceptance of a state. To divert a transition, we select a uniformly random transition, and change its destination state to a uniformly random state. To add a state, we add a new state with random acceptance, and divert a random transition to it. To remove a state, we uniformly select a random state to remove, and divert its reaching transitions to other states. Finally we select a random existing state to flip its acceptance too.

The second family of experiments, which we call the *Feature Benchmark* aims to emulate a more realistic scenario: extending an existing software with a new feature, or new behavior. We do this by introducing a small (again canonical) *feature automaton* of 3 states to the t_0 automaton by diverting 3 random transitions into the start state of the feature automaton.

As the automata we consider, including the feature automata are all minimal, we guarantee that all the behavior in our t_{10K} targets is observable and reachable. The t_{10K} targets may naturally not be minimal anymore, however we believe this to be realistic: System evolution makes no promises on some kind of "monotonic growth", it is only reasonable to assume that certain areas of the system remain unchanged. Additionally, this means that although our (non-minimized) state space increases, the Myhill–Nerode equivalence relation may actually shrink, or change completely.

4.3.2.2 Answering MAT Queries

As mentioned in Section 2.3, most classic and adaptive automata learning algorithms work in the *minimally adequate teacher* framework, where they can pose membership and equivalence queries. Answering membership queries is usually straightforward and done through interaction with the target system. However, answering equivalence queries is, in practice, the most expensive part of learning [Vaa17; ATW24]. Realizing equivalence queries in closed-box learning is usually done through approximate testing, meaning that answering one equivalence query is done by performing *many* membership queries, probing the system for different behavior. Many sophisticated algorithms have been developed for efficiency closed-box testing, such as the Wp-Method [Fuj+91], however, for simplicity, and as to not bias our results to the advantages of specific complex testing algorithms, in our experiments we answer

equivalence queries through random probing.

Specifically, we generate random words in $\Sigma^\star$ of varying length, and test the target system against a learner hypothesis in search for discrepancies. Each word is formed by, starting from the empty word, with 90% probability appending an extra uniformly random symbol, or terminating the word. These words are then tested until a counterexample is found, or we hit a limit number of test words, which for our case we set to 10,000. For the state and alphabet sizes we consider, this bound proved to be more than sufficient.

Additionally, as is commonly the case with automata learning applications, such as PROGNOSIS of Chapter 3, we introduce a *cache layer* at the query level, meaning that if the same word is queried twice, we only incur (and count) this cost once. Naturally, at the point that the target t_0 mutates into t_{10K}, we reset the cache.

This then allows us to have a consistent measure of *cost* in learning: the number of system queries, or interactions used in learning, be it in answering membership or equivalence queries. We use the number of queries posed then as the "clock tick" of the learning algorithms, and mutate the target t_0 into t_{10K} at 10K queries. For the classic KV algorithm, as it cannot learn evolving systems at all, we have to run the algorithm twice, first targeting t_0, then from scratch targeting t_{10K}. To ensure repeatable results, each data point in question has been run 300 times, each with fresh random seeds of the same parameters, and averaged.

4.3.2.3 Quantifying Learning Progress

In our analysis of the difference in behavior between classic and adaptive learning algorithms, it is useful to have a fine grained notion of *progress*. We understand that classic algorithms always start from knowing nothing about the target, and terminate with a perfect model, and that adaptive learning algorithms start with "some" understanding of the target, and terminate with a perfect model. However, it would be better if we could formalize this notion of starting from "some" understanding of the system, so that we can visualize differences in behavior. This allows us to see how quickly adaptive learners recover from changes in the target, compared to their classic counterparts. We do so with the following definition of progress.

Definition 4.3.1. *Given $\alpha \in (0,1)$, a target automaton t_i, and a hypothesis automaton h_i, the* progress p_i *is given by:*

$$p_i = \sum_{\substack{w \in \Sigma^\star, \\ [\![t_i]\!](w) = [\![h_i]\!](w)}} (1-\alpha) \cdot \left(\frac{\alpha}{|\Sigma|}\right)^{|w|}$$

△

The *progress* is then a measure of the alignment between target and hypothesis DFAs on the (infinite) space of possible words $\Sigma^\star$. The parameter α controls how much weight is placed on longer words versus shorter ones. When α is close to 0, nearly all the mass sits on the empty word; when α is close to 1, the mass spreads out to longer and longer words. As such, α allows us to tune an "inspection focus", determining how far into the behavior of the automata we want to distribute most of the weight. Importantly, α governs only the spread of this mass, however does not impact two key properties: p_i operates on the unit interval, and the extremes of the scale represent full correctness (for $p_i = 1$), and full incorrectness (for $p_i = 0$).

Lemma 4.3.2. *For all $\alpha \in (0,1)$, and $i \in \mathbb{N}$, $0 \leq p_i \leq 1$.*

Proof. Each summand $(1-\alpha) \cdot (\alpha/|\Sigma|)^{|w|}$ is strictly positive for $\alpha \in (0,1)$, so $p_i \geq 0$. For the upper bound, the agreement set is a subset of $\Sigma^\star$, therefore:

$$\begin{aligned}
p_i &\leq \sum_{w \in \Sigma^\star} (1-\alpha) \cdot \left(\frac{\alpha}{|\Sigma|}\right)^{|w|} && \text{(sum over all words)} \\
&= (1-\alpha) \sum_{n=0}^{\infty} \sum_{w \in \Sigma^n} \left(\frac{\alpha}{|\Sigma|}\right)^{w} && \text{(regroup by word length)} \\
&= (1-\alpha) \sum_{n=0}^{\infty} |\Sigma|^n \cdot \frac{\alpha^n}{|\Sigma|^n} && (|\Sigma|^n \text{ words, each weighted } \alpha^n/|\Sigma|^n) \\
&= (1-\alpha) \sum_{n=0}^{\infty} \alpha^n && \text{(cancel } |\Sigma|^n) \\
&= (1-\alpha) \cdot \frac{1}{1-\alpha} && \text{(geometric series)} \\
&= 1
\end{aligned}$$

□

Lemma 4.3.3. *For all $\alpha \in (0,1)$, and $i \in \mathbb{N}$:*

$$p_i = 1 \iff \forall w \in \Sigma^\star.\ [\![h_i]\!](w) = [\![t_i]\!](w)$$
$$p_i = 0 \iff \forall w \in \Sigma^\star.\ [\![h_i]\!](w) \neq [\![t_i]\!](w)$$

Proof. Let $A_i = \{w \in \Sigma^\star : [\![h_i]\!](w) = [\![t_i]\!](w)\}$ and $D_i = \Sigma^\star \setminus A_i$. Since A_i and D_i partition $\Sigma^\star$, we have:

$$\begin{aligned} p_i &= \sum_{w \in A_i} (1-\alpha) \cdot \left(\frac{\alpha}{|\Sigma|}\right)^{|w|} && \text{(definition of } p_i\text{)} \\ &= 1 - \sum_{w \in D_i} (1-\alpha) \cdot \left(\frac{\alpha}{|\Sigma|}\right)^{|w|} && \text{(previous lemma)} \end{aligned}$$

($p_i = 1$). ($\Leftarrow$) If $A_i = \Sigma^\star$ then $D_i = \varnothing$, so $p_i = 1 - 0 = 1$. ($\Rightarrow$) If $p_i = 1$ then $\sum_{w \in D_i} (1-\alpha) \cdot (\alpha/|\Sigma|)^{|w|} = 0$. Since each summand is strictly positive for $\alpha \in (0,1)$, the sum is zero only if $D_i = \varnothing$, i.e. $A_i = \Sigma^\star$.

($p_i = 0$). ($\Leftarrow$) If $A_i = \varnothing$ then p_i is an empty sum, so $p_i = 0$. ($\Rightarrow$) If $p_i = 0$ then $\sum_{w \in A_i} (1-\alpha) \cdot (\alpha/|\Sigma|)^{|w|} = 0$. Again, each summand is strictly positive, so the sum is zero only if $A_i = \varnothing$, i.e. $D_i = \Sigma^\star$. □

For our experiments we will be using $\alpha = 0.999$. At this value, even disagreement at words of length ≈ 500 contribute non-negligible weight, and this allows for more visual contrast when comparing algorithms below. To actually *compute* the progress values exactly despite the infinite summation, we instead perform this computation at the automaton level, where we determine the progress value for each pair of states, relative to the progress of successor states. This dependency forms a linear system which we solve through Gaussian elimination.

Proposition 4.3.4. *For all $\alpha \in (0,1)$ and $i \in \mathbb{N}$, the progress p_i is the unique solution $v_{(q_0,r_0)}$ of the linear system over state pairs $(q,r) \in Q_{t_i} \times Q_{h_i}$:*

$$v_{(q,r)} - \frac{\alpha}{|\Sigma|} \sum_{a \in \Sigma} v_{(\delta(q,a),\delta(r,a))} = \begin{cases} 1-\alpha & \text{if } F_t(q) = F_h(r) \\ 0 & \text{otherwise.} \end{cases}$$

Proof. For each pair (q,r), define:

$$v_{(q,r)} = \sum_{\substack{w\in\Sigma^\star,\\ [\![t_i]\!](q)(w)=[\![h_i]\!](r)(w)}} (1-\alpha)\cdot\left(\frac{\alpha}{|\Sigma|}\right)^{|w|}$$

so that $p_i = v_{(q_0,r_0)}$ by definition. Below, we use *Iverson brackets* $[p]$ such that $[p]=1$ if $p=\top$, and $[p]=0$ otherwise. We will use the partition of $\Sigma^\star$ into $\{\varepsilon\}$ and $\{aw : a\in\Sigma, w\in\Sigma^\star\}$. We then have that:

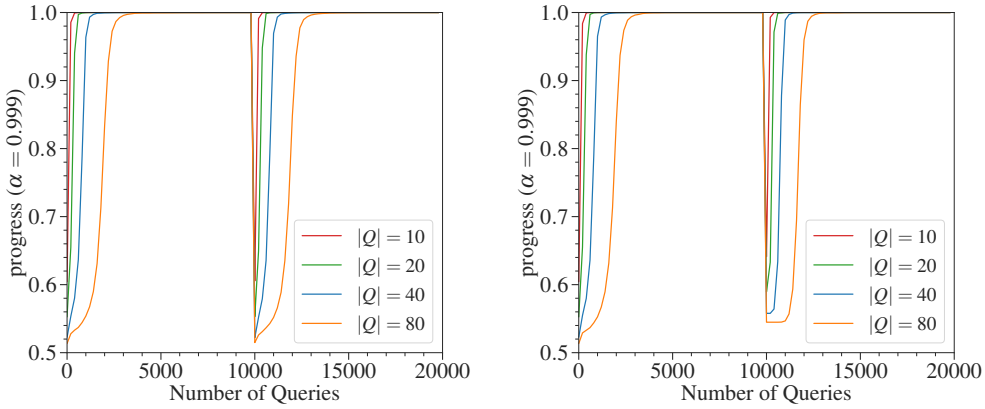
$$\begin{aligned}
v_{(q,r)} &= \sum_{\substack{w\in\Sigma^\star,\\ [\![t_i]\!](q)(w)=[\![h_i]\!](r)(w)}} (1-\alpha)\cdot\left(\frac{\alpha}{|\Sigma|}\right)^{|w|} && \text{(definition)}\\
&= [F_t(q)=F_h(r)]\cdot(1-\alpha) + \sum_{a\in\Sigma}\sum_{\substack{w\in\Sigma^\star,\\ [\![t_i]\!](q)(aw)=[\![h_i]\!](r)(aw)}} (1-\alpha)\cdot\left(\frac{\alpha}{|\Sigma|}\right)^{|aw|}\\
& && \text{(partition into } \varepsilon \text{ and } a\cdot\Sigma^\star)\\
&= [F_t(q)=F_h(r)]\cdot(1-\alpha) + \sum_{a\in\Sigma}\sum_{\substack{w\in\Sigma^\star,\\ [\![t_i]\!](\delta(q,a))(w)=[\![h_i]\!](\delta(r,a))(w)}} (1-\alpha)\cdot\left(\frac{\alpha}{|\Sigma|}\right)^{|w|+1}\\
& && \text{(unfold transitions)}\\
&= [F_t(q)=F_h(r)]\cdot(1-\alpha) + \frac{\alpha}{|\Sigma|}\sum_{a\in\Sigma}\sum_{\substack{w\in\Sigma^\star,\\ [\![t_i]\!](\delta(q,a))(w)=\\ [\![h_i]\!](\delta(r,a))(w)}} (1-\alpha)\cdot\left(\frac{\alpha}{|\Sigma|}\right)^{|w|}\\
& && \text{(factor out } \alpha/|\Sigma|)\\
&= [F_t(q)=F_h(r)]\cdot(1-\alpha) + \frac{\alpha}{|\Sigma|}\sum_{a\in\Sigma} v_{(\delta(q,a),\,\delta(r,a))} && \text{(definition of } v)
\end{aligned}$$

Rearranging gives the stated system, which in matrix form is $Av=b$ where $A = I - \frac{\alpha}{|\Sigma|}M$. M is the degree matrix of the product automaton, with $M_{(q,r),(q',r')} = |\{a\in\Sigma : \delta(q,a)=q' \wedge \delta(r,a)=r'\}|$. Each row of M sums to $|\Sigma|$, so each row of A sums to $1-\alpha>0$, making A strictly diagonally dominant and therefore invertible. The system thus has a unique solution.

□

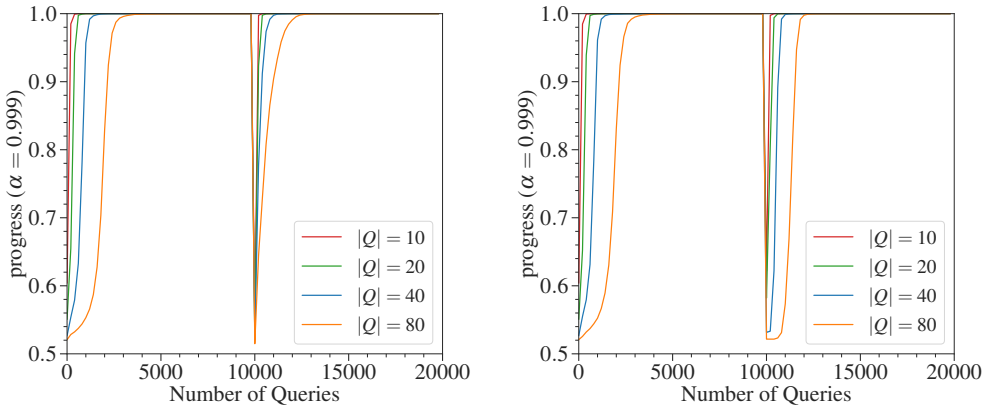
4.3.3 Results and Analysis

We can now analyze the results of our experiments, and consider their implications for our research questions. We start by plotting the average progress when learning each benchmark through our IKV algorithm. As reference, we plot the same for the classic KV algorithm so that we can highlight differences in behavior between the classic and incremental variants. We then move to focusing on how quickly different algorithms recover from evolution in the target, considering IKV, the classic KV baseline, and the PDL* algorithms.



(a) Classic KV algorithm. **(b)** Incremental IKV algorithm.

Figure 4.1: Mutation Benchmark: Average learning progress graphs.



(a) Classic KV algorithm. **(b)** Incremental IKV algorithm.

Figure 4.2: Feature Benchmark: Average learning progress graphs.

In Figures 4.1b and 4.2b we can see the average learning progress graphs for the IKV algorithm on the Mutation and Feature benchmarks, respectively. For reference

we plot the baseline KV progress in Figures 4.1a and 4.2a. The x-axis determines the number of queries used so far, and the y-axis the progress at such point. I.e., the progress of the hypothesis h_i produced up to the given number i of queries used, relative to the relevant target automaton t_i. Each line reflects a different target pair in the family, of increasing state sizes from 10 to 80 states. The "faster" an algorithm produces hypotheses with progress values of 1.0, in terms of query cost, the better.

We can see that both algorithms have learning runs that start from around the same progress: just above 0.5. This is expected, as both the KV and IKV algorithms start the run with a single-state hypothesis, that is either accepting or rejecting depending on the empty word. As our progress measure actually forms a probability distribution over $\Sigma^\star$, the set of words, it is then reasonable that a first hypothesis is just slightly better than a 50-50 chance of a given word being correctly classified. Once the target evolves (at 10K queries), we can see that the classic algorithm's progress drops to about the same progress level as the start, while the incremental algorithm drops to a progress level not as low. This is because while the KV algorithm must start the second half of the run from scratch, due to its inability to reuse previous knowledge, the incremental algorithm is starting from a minimized previous hypothesis, where outdated states are pruned according to the new target.

Comparing more broadly, the halves of the progress lines for the classic KV algorithm are very similar. They may not have the same gradients due to the t_0 and t_{10K} targets being different, but importantly they converge to 1.0 using a similar number of queries. The incremental algorithm, on the other hand, has a t_0 run very similar to the classic algorithm, due to the lack of previous knowledge, but the t_{10K} segment is already very different.

Immediately we see that it is not always the case that the IKV progress line ascends immediately at the t_{10K} evolution point. This is because, as we are not starting from a widely dissimilar automaton, the equivalence testing implementation actually requires a noticeable number of test queries to find a counterexample. Nonetheless, this is overpowered by a generally faster convergence to 1.0 progress, compared to the t_0 half of the run.

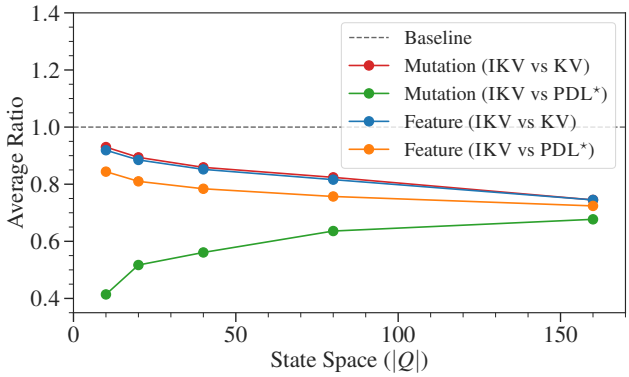


Figure 4.3: Ratio of number of queries since 10K needed to fully learn the t_{10K} target.

These learning progress graph representations are good for demonstrating the overall behavior and approach of the learning algorithm, but we now want to evaluate whether this algorithm does indeed provide a benefit over learning systems classically, or with the state-of-the-art PDL* algorithm, in terms of the number of queries it takes them to fully recover from evolutions in the target. We can do this by considering, for each run, the number of queries needed to go from point of evolution, to the first hypothesis with 1.0 progress. This information is naturally present in the progress graphs, however it is better seen through a comparison of the *ratio* in queries needed between the IKV algorithm, and the KV or PDL* algorithms.

In Figure 4.3 we plot such ratio, for the targets of increasing state space considered, in both the Mutation and Feature benchmarks, against each of the competing algorithms. A ratio of x indicates that the IKV algorithm required x times the queries to recover from the mutation and reach the correct new target, compared to the other algorithm. The lower the ratio is from 1, the better the relative performance of IKV.

As we can see, the IKV algorithm consistently outperforms the algorithms in all benchmarks, even with small targets where a relatively big portion of the system has changed. When compared to PDL*, we can see that the IKV algorithm still consistently outperforms in terms of queries, with a tendency to plateau between a 0.75 and 0.7 ratio: a reduction by 25–30%. We note that this is not just the effect of using a more efficient data structure like the classification tree: even compared to classic KV, our IKV algorithm provides significant improvement, with a reduction in the number of queries tending to 20–25%.

4.4 Related Work

Our algorithm contributes to the field of adaptive learning first introduced by Groce et al. [GPY02] with the development of *adaptive* model checking. The idea is to use adaptive automata learning in producing new models, so that the new states found can be model checked first. This was done by slightly modifying the $L^\star$ algorithm to start from a previous set of access sequences. Chaki et al. [Cha+08] use a similar algorithm in combination with assume-guarantee reasoning [Jon83] to provide a framework where model checking is used to find counterexamples in the model, and thus make progress in learning. The algorithm of Groce et al. [GPY02], Dynamic $L^\star$, reuses not only the starting prefix/suffix sets, but also their computed values. However, these must still be validated on the new target. Finally, Damasceno et al. [DMD19] introduced Partial Dynamic $L^\star$ ($PDL^\star$), which improves Dynamic $L^\star$ by analyzing the start prefix/suffix sets to trim them where possible, reducing the amount of information that needs to be validated.

These previous algorithms, however, suffer from using an observation table as their data structure, which increases the amount of redundant data to be acquired. As the tables grow, these redundancies significantly increase the number of queries that need to be posed, both in learning and in rechecking the old information.

More recently, and postdating the publication of this chapter's work, the Adaptive $L^\sharp$ ($AL^\sharp$) incremental learning algorithm has been contributed by Kruger et al. [KJR25]. The $AL^\sharp$ algorithm is an incremental version of the $L^\sharp$ algorithm [Vaa+22], now one of the leading state-of-the-art classic automata learning algorithms, developed also after the publication of this work. It works by operating on an observation tree, which is itself already a *bona fide* automaton. This provides a more direct connection between managing the information gathered about the target so far, and turning such structure into a respective hypothesis. The $AL^\sharp$ algorithm extends this by being able to also reuse a past learned automaton, whose states are then matched with the new observation tree being learned. This "corrected" structure is then extended classically like with $PDL^\star$ and IKV.

4.5 Discussion

We introduced IKV, a new algorithm for *adaptive learning* which provides the (at the time of publication) most efficient adaptive learner, allowing us to learn systems that were before too big or evolved too quickly to be learned classically. We evaluated the fitness of this algorithm through a set of realistic experiments, demonstrating their benefit over classic and existing adaptive learners. Adaptive learning could provide a cooperative relationship between active and passive learning, due to its flexibility towards information that changes over time and becomes available at different stages. We want to explore this in future work, as well as developing a formal framework of adaptive learning, where other algorithms can be easily adapted, e.g., efficient classic algorithms such as TTT [IHS14]. Finally, although our incremental algorithm allows us to learn systems that evolve at known points, this would not work in a true closed-box scenario, where we cannot know if or when the system changes. In the future we would like to then develop a *continuous* learning framework for such evolving systems. In Chapter 5 we consider contributions towards this goal.

Chapter 5

Conflict-Aware Active Automata Learning

The MAT learning framework due to Angluin [Ang87] has allowed the development of provably correct, terminating learning algorithms for automata such as DFAs and Mealy machines, as well as more complex models [Cas+16; Moe+17; DD17; Van+20; VEB23]. However, when applying these algorithms to true closed-box systems, the equivalence query — the cornerstone of their theoretical guarantees — is known to be unrealizable in general [PVY99; Vaa17].

Even when interactions are restricted to membership queries, real-world learning is rarely foolproof: interactions with the System Under Learning (SUL) can be *noisy*, as in PROGNOSIS's interactions with network implementations subject to inconsistent packet-loss (Chapter 3). Such noise leads to *conflicting observations* — different outputs for the same inputs — which classic MAT-style learners cannot reconcile.

In this chapter we introduce the Conflict-Aware Active Automata Learning framework (CƐAL, pronounced *seal*): an alternative formalization of the learning framework that automata learning algorithms operate under, focused on realistically modeling the learning environment to improve the success and efficiency of automata learning in the face of conflicting observations. Any existing MAT learner can be used in CƐAL: when a conflict arises, we provide a method for updating the learner's internal state — without assumptions on its data structure — so that it remains conflict-free while removing only inconsistent information. In this way, classic MAT

learners gain a way to recover from conflicts caused by, e.g., noise in interactions or evolution of the system at unobserved points.

At the heart of CƐAL is the use of an *observation tree*, a data structure external to the learner that stores information gathered from the SUL. The learner uses the tree to construct its own internal data structure, and when a conflict appears, the tree is updated while the learner's data structure is reduced to a conflict-free point and then expanded from the tree. Crucially, the learner reuses observations already stored in the tree, without requiring fresh tests on the SUL for previously observed behaviors. CƐAL's main features are:

– The SUL is a first-class citizen rather than an abstraction. CƐAL notably does not rely on *equivalence queries*, replacing them with either a check of the stored knowledge (when sufficient) or an *equivalence test* using an off-the-shelf testing algorithm (e.g., the Wp-method [Fuj+91] or Hybrid-ADS [LY94; Sme+15]).

– Observations on the SUL are stored in an observation tree managed by a new Reviser agent, which handles conflicts and answers the learner's queries like a teacher. This teacher interface enables the use of any MAT-based algorithm seamlessly, requiring only the ability to restart a classic MAT learner.

– The Reviser alone interacts with the SUL, issuing observations to expand its tree.

Crucially, CƐAL is less abstract than MAT, representing directly the objects and challenges of *practical* active learning, while still letting algorithm designers enjoy the simplifying abstraction of MAT.

The remainder of this chapter is structured as follows. Section 5.1 formalizes the notion of *conflict* in observations and shows how it models real-world perturbations that jeopardize the applicability of automata learning. Section 5.2 presents the CƐAL framework and details the Reviser agent. Section 5.3 evaluates CƐAL using a broad range of automata learning benchmarks [Nei+19], comparing several state-of-the-art algorithms on targets of varying size and noise levels, while varying the controllable parameters for both MAT and CƐAL. The results show that under noise, CƐAL drastically reduces the number of observations required to learn correct models, and

improves the success rate from MAT's 79.5% to 95.5%. Section 5.5 concludes with an overview of related and future work.

5.1 Conflicts in Automata Learning

We now characterize the mode of interaction with the SUL in the CƐAL framework, and motivate our model of *conflict* in observations with realistic behavior. As further detailed in Section 5.2, CƐAL does not hide the SUL behind a teacher providing membership or equivalence queries, but rather considers directly the interactions with the SUL as the canonical way of modeling the system. The CƐAL framework is tailored to learning Mealy machines (cf. Definition 2.2.10), as these tend to be the central model for representing the input-output behavior of reactive systems[1]. The unit of interaction with the SUL is then the input-output pair recorded from a test:

Definition 5.1.1 (Observation)**.** *An* observation *is a pair of words* $(i,o) \in \Sigma^\star \times \Gamma^\star$ *obtained by sending the input i to the SUL and receiving the output o.* △

Under a strictly abstract setting such as MAT, an observation (i,o) would imply that the target Mealy machine $\mathcal{M}$ is such that $[\![\mathcal{M}]\!](i) = o$. However, in practice we must account for perturbations in the environment that might affect observations. These can be, for instance, in the form of noisy observations, or evolutions of the target system that make some observations outdated. Not only can this affect the quality of the model produced, noise and evolutions can lead to *conflicts*, where two pieces of information cannot possibly hold in conjunction given the semantics of the Mealy machine being learned.

Definition 5.1.2 (Conflict)**.** *Two observations* (i,o) *and* (i',o') conflict*, written* $(i,o) \not\downarrow (i',o')$*, when there is an input word* $i'' \in \mathsf{prefixes}(i) \cap \mathsf{prefixes}(i')$ *such that* $o[0..|i''|-1] \neq o'[0..|i''|-1]$. △

Relying on a notion of conflict that is determined by the expressive limits of Mealy machines allows our framework to be flexible enough to cover many sources of conflict when applying automata learning to real-world systems.

[1] Nonetheless, the core working principles of CƐAL can be adapted to other models such as DFAs.

5.1.1 Observation Noise

The term *noise* is usually used to describe a wide range (if not any form) of perturbations that can happen between the controlled agent (in our case the Learner) and the SUL. In the case of this study we are primarily interested in the classification between *input* and *output* noise:

Output noise We call output noise a perturbation that only affects what our agent sees from the world, i.e., the outputs of the SUL. Formally, this kind of noise can be represented as a probabilistic version of the semantics $[\![\mathcal{M}]\!]$ of the target, where $[\![\mathcal{M}]\!](i)$ returns a different output word caused by potentially changing any of the symbols in the real output.

Input noise This kind of noise affects the observation input i sent to the SUL, so that a different input word i' of same size is processed instead.

Example 5.1.3 (Packet-Loss in PROGNOSIS)**.** *The* PROGNOSIS *tool of Chapter 3 learns models of network protocol implementations by interacting with them over a network. Networks are lossy channels susceptible to congestion, and link faults that can lead to* packet-loss*, where a given packet is sometimes not received. As such, we could have the following observations from a single 3 symbol input* [SYN ACK PSH]*:*

([SYN ACK PSH], [SYN-ACK NIL PSH])	*successful handshake,* PSH *exchange.*
([SYN ACK PSH], [NIL SYN-ACK NIL])	*successful handshake, return* PSH *lost.*
([SYN ACK PSH], [NIL NIL RST])	*unsuccessful handshake, input* SYN *lost.*

While the first observation leads to a successful handshake and data transmission (noise-free), the second observation fails to receive data due to the missed return PSH *packet (output noise), and the third observation fails to handshake due to a lost input* SYN *packet (input noise). All three observations conflict with each other. Indeed, (deterministic) Mealy machines are not capable of producing different outputs for matching inputs.*

Noise can have different levels of *structure*, being generated by different kinds of models or probability distributions. As this chapter strives for a generic approach, no assumption is made on the structure of noise. One notable restriction of our approach is that it does not target adversarial modifications, such as an attacker trying to change the Learner's hypotheses by carefully mutating symbols in-transport in a non-conflicting way.

5.1.1.1 Mitigations

The standard mitigation for noise in active automata learning is to repeat each query multiple times and majority-vote the result. This strategy is grounded in Angluin and Laird [AL88]'s *classification noise model*, which establishes that correct identification is possible if and only if the noise rate satisfies $\eta < 1/2$ — i.e., the correct answer is the most frequently occurring outcome. Sakakibara [Sak91] extends this result to the query setting, showing that $O(\frac{1}{(1-2\eta)^2} \log \frac{1}{\delta})$ repetitions per query suffice to recover the correct answer with probability $1 - \delta$. This is the theoretical basis for the repeat-based noise filtering used in practice by automata learning applications such as PROGNOSIS and its related work (Section 3.6). However, correctly calibrating the number of repeats is difficult in practice: the noise rate η is generally unknown and inconsistent across the learning run. Underestimating leads to noisy observations corrupting the learner's data structure, causing incorrect hypotheses, or fatally compromising the run in the case of conflicts. Overestimating wastes queries, risking timeouts before learning completes with a correct model.

CεAL addresses this by *relaxing the requirement that all noise be filtered before reaching the learner*. By tolerating some noise leaking through the sampling filter and handling the resulting conflicts at the framework level, CεAL can operate effectively at significantly lower observation repeats than MAT requires. The experiments of Section 5.3 demonstrate that this yields both higher success rates (95.5% vs. 79.5%) and fewer total observations across a broad range of noise rates and targets.

5.1.2 System Evolutions

Another source of conflicting observations now familiar to us is *evolutions* of the target SUL itself. Unlike noise, which perturbs individual observations while the underlying system remains fixed, a system evolution permanently alters the target, rendering previously correct observations outdated. Formally, we can think of an evolution in a target Mealy machine $\mathcal{M}$ as its observations being answered instead by a different Mealy machine $\mathcal{M}'$.

We consider three types of system evolutions, in increasing learning difficulty:

Additive Evolutions An additive evolution happens when the target mutates in a strictly non-conflicting way with respect to the inputs already queried by the learner. For instance, the system's alphabet is extended to perform some new behavior previously unlearned, or the system changes in a way that strictly refines the existing Myhill–Nerode classes further without invalidating any previously observed behavior. Formally, let $O \subseteq \Sigma^\star$ be the finite set of inputs queried by the learner so far. Then:

$$\nexists i \in O.\ (i, [\![\mathcal{M}]\!](i)) \not\sim (i, [\![\mathcal{M}']\!](i)).$$

Open-Box Evolutions These evolutions happen when the system changes in way that may introduce conflicting behavior, however the system reveals (or even allows control over) *when* an evolution happens.

Closed-Box Evolutions Finally, closed-box evolutions happen when the system changes in a potentially conflicting way, however no notification or control is given of when such an evolution happens. The system will simply start behaving differently for the unknown section that has mutated.

5.1.2.1 Mitigations

The difficulty of handling system evolutions during learning depends critically on how much information the learner has about the change.

Additive evolutions are the simplest case: since no existing observation is invalidated, the learner's data structure remains sound. A standard MAT algorithm

will detect the evolution naturally when its current hypothesis fails an equivalence check against the (now extended) system, producing a counterexample that witnesses the new behavior. The learning loop then proceeds as usual, refining the hypothesis to accommodate the addition. No special machinery is required beyond the normal counterexample-driven refinement loop.

Open-box evolutions are more challenging, as previously collected observations may now conflict with the mutated system. However, because the learner knows *when* the evolution occurred, it can identify which observations predate the change and selectively invalidate them. This is precisely the setting targeted by adaptive automata learning algorithms such as the incremental Kearns–Vazirani algorithm of Chapter 4, which maintains a classification tree that can be surgically updated when the target changes at a known point. The key working mechanism is that the learner's intermediate computations, such as counterexample analyzes, state discriminations, and hypothesis constructions, remain sound up to the moment of evolution, and can be cleanly separated from the post-evolution phase.

Closed-box evolutions, however, are fundamentally out of reach of current adaptive automata learning. Without knowledge of when the evolution occurred, the learner faces a compound problem: not only must it detect that the system has changed (by observing conflicts with previously consistent data), but it cannot determine which of its stored observations reflect the current system and which are outdated. Furthermore, there is no guarantee that the system will only evolve once a given learning run finishes. The system could mutate at critical points, such as counterexample analysis or hypothesis construction. This can lead the algorithm to completely corrupt the invariants of its data structure.

Unlike additive evolutions, simply restarting the learning loop does not suffice, as the learner's data structure is silently corrupted. Unlike open-box evolutions, there is no clean temporal boundary at which to prune observations.

5.2 The Framework

Irrespective of the nature of the conflicts detected, dealing with them requires the ability to *backtrack* certain decisions that were made based on what is now considered incorrect information. This pinpoints an issue with most MAT learners: There is no notion of information storage other than the internal data structure that the learners use to build the model, which is not easily updated in the face of conflict.

We propose instead our alternative to MAT in practice, the *Conflict-Aware Active Automata Learning* framework. CƐAL's main features are as follows:

– The SUL is a first class citizen, allowing for clearer practical discussions on modularity and managing conflicts arising in its interaction.

– The information obtained through observations on the SUL is stored in a new *Reviser* agent that resolves the conflicts, and answers the learner's queries in an MAT interface like a teacher would.

– The Reviser alone interacts with the SUL by means of observations meant to expand its data structure, the observation tree. The learner's queries are then answered directly from this observation tree.

5.2.1 Overview

CƐAL (Figure 5.1) is centered around three agents: the Learner, the System (SUL), and the Reviser, together with the interfaces between them.

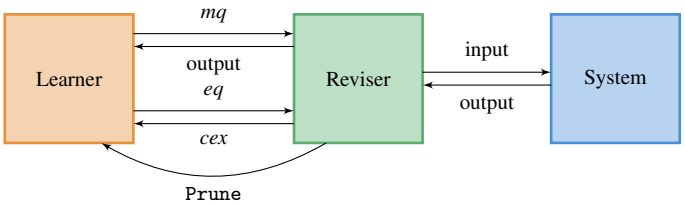


Figure 5.1: Simplified view of the CƐAL framework. See Figures 5.3 and 5.4 for details.

The Learner plays the same general role as in MAT. Crucially, any MAT learner can be used in CƐAL (e.g., $L^\star$, KV, TTT, $L^\sharp$). The Learner *does not have* to store the information obtained from observations on the system. It focuses on the questions

"What is the next query to make?" and "How is the hypothesis built?". The System is the System Under Learning, together with its environment (e.g., noise). The Reviser handles knowledge and conflicts. It answers the question "What do we know about the system?", and is set between the Learner and the System with interfaces to both.

Remark 5.2.1. *The Reviser acts as a MAT teacher w.r.t. the Learner, answering membership and equivalence queries, with the added ability to* `Prune` *the learner to place it in a state coherent with the Reviser's information. On the System's view, the Reviser acts as a tester, providing observation input sequences and recording the system output. The views of the learner and system in* CƐAL *are illustrated below.*

(a) Learner's view **(b)** System's view

Figure 5.2: The Reviser's interfaces.

The *interfaces* on the Learner side are similar to MAT: the Learner can perform membership queries (*mq*) and equivalence queries (*eq*) on the Reviser, with the latter potentially resulting in a counterexample (*cex*). Queries are sent to the Reviser and not directly to the SUL, allowing us to control the information that the Learner obtains, and reuse the information in the Reviser with no new observations. Formally, CƐAL provides the following functions as module interfaces:

– $mq : \Sigma^\star \to (\Gamma^\star + \{\texttt{Prune}\})$ the membership query of the learner that the Reviser has to implement. It varies from the MAT function as the Reviser may return a command to prune the learner's state instead of an output.

– $eq : \mathsf{Mealy} \to (\Sigma^\star + \{\texttt{Prune}\})$ the equivalence query of the learner that the Reviser has to implement. It may return `Prune` if a conflict is detected.

– $sys : \Sigma^\star \to \Gamma^\star$ is a call to the system for a specific input. The system returns the corresponding output, with any potential noise or other conflicts present.

In the interface mentioned above, an *eq* can never return no counterexample as in MAT. This work is left to the Reviser, which halts the learning process according to the chosen termination criterion. The `Prune` signal does not require us to modify an existing MAT Learner: upon receiving it, the current learner execution is aborted and the Learner is reset to the initial state of a fresh execution of the same learning algorithm, discarding all learner-local state accumulated so far. The Reviser and its observation tree are preserved, so the restarted Learner can reconstruct its state from the observations that remain valid. Since these observations are served from the Reviser's cache, this operation does not add to the observation use of the run.

5.2.2 The Reviser

The Reviser agent is the core of the CℰAL framework. It concretizes its main idea: taking the storage and handling of observations *out* of the Learner's prerogatives. Its task is to update its internal data structure, the observation tree, on which a Learner is trained. Formally, an observation tree is a tree-shaped *partial* Mealy machine storing a finite set of observations.

Definition 5.2.2 (Partial Mealy Machine)**.** *A* partial *Mealy machine is a tuple* $\mathcal{M} = \langle Q, q_0, \Sigma, \Gamma, \delta, \lambda \rangle$ *where:* Q *is a finite set of states;* q_0 *is the* initial *state;* Σ *is a finite* input *alphabet;* Γ *is a finite* output *alphabet;* $\delta \colon Q \times \Sigma \to (Q + \mathbb{1})$ *is a transition function; and* $\lambda \colon Q \times \Sigma \to (\Gamma + \mathbb{1})$ *is the* output *function. The semantics of* $\mathcal{M}$ *is given by the function* $[\![\mathcal{M}]\!](q_0) \colon \Sigma^\star \to (\Gamma^\star + \mathbb{1})$ *defined by:*

$$[\![\mathcal{M}]\!](q)(\varepsilon) \triangleq \varepsilon \quad [\![\mathcal{M}]\!](q)(a \cdot w) \triangleq \begin{cases} o \cdot v & \text{if } \lambda(q,a) = o \in \Gamma, \delta(q,a) = q' \in Q, \\ & [\![\mathcal{M}]\!](q')(w) = v \in \Gamma^\star \\ \bullet \in \mathbb{1} & \text{otherwise} \end{cases}$$

Where clear, we omit the initial state parameter $[\![\mathcal{M}]\!](q_0)$*, writing only* $[\![\mathcal{M}]\!]$*.* △

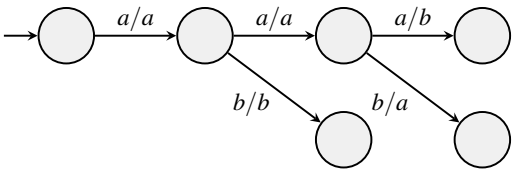
For convenience, we define also the *language* of a partial Mealy machine to be the finite set of observations it produces.

Definition 5.2.3. *Given a partial Mealy machine $\mathcal{M}$ we define its* language *by:*

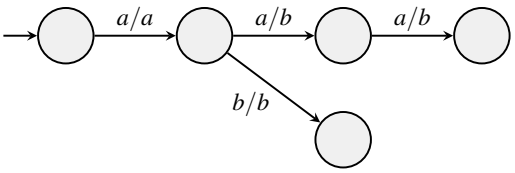
$$\mathcal{L}_{\mathcal{M}} \triangleq \{(i,o) \mid i \in \Sigma^\star, [\![\mathcal{M}]\!](i) = o \in \Gamma^\star\}$$

△

Example 5.2.4. *Below we depict observation trees over alphabets $\Sigma,\Gamma \triangleq \{a,b\}$, and their respective languages.*



$\mathcal{L} = \{(aaa,aab),(aab,aaa),(ab,ab)\}$



$\mathcal{L} = \{(aaa,abb),(ab,ab)\}$

The Reviser agent is then responsible for maintaining an observation tree that maximally captures the non-conflicting observation data seen so far. When a new observation is received, the Reviser updates the tree to hold this knowledge, taking precedent over older conflicting information, pruning it if needed. In the case that a conflict is detected, it is the Reviser's role to send a `Prune` command to the learner.

Remark 5.2.5. *We have opted for a* most recent *update strategy of the Reviser's observation tree, where in order to resolve a conflict, the most recent non-conflicting observation data will prevail. One could develop alternative strategies, such as selecting most frequent, or stable information, depending on specifics of the source of conflict. However, our preliminary investigation found that the most recent update strategy is effective for the sources of conflict we consider (see Section 5.1), and our experimental evaluation (Section 5.3) supports the use of this update strategy.*

Definition 5.2.6. *The* empty observation tree *is defined $\mathcal{T}_0 \triangleq \langle Q, q_0, \Sigma, \Gamma, \delta, \lambda \rangle$ with:*

$$Q \triangleq \{q_0\} \qquad \delta(q_0, a \in \Sigma) \triangleq \bullet \in \mathbb{1} \qquad \lambda(q_0, a \in \Sigma) \triangleq \bullet \in \mathbb{1}$$

△

Definition 5.2.7 (Reviser)**.** *A* Reviser *agent is initialized with an empty observation tree $\mathcal{T}$ and implements four operations from it:*

– $\texttt{read}_{\mathcal{T}}\colon \Sigma^\star \to (\Gamma^\star + \{\texttt{Prune}\})$ *looks in* $\mathcal{T}$ *for an observation and returns it or tests the system if needed. If a test is performed, the observation is written to* $\mathcal{T}$.

– $\texttt{write}_{\mathcal{T}}\colon (\Sigma^\star \times \Gamma^\star) \to (\{\texttt{Prune}\} + \mathbb{1})$ *updates* $\mathcal{T}$ *with the observation gained from a system interaction. It optionally returns* `Prune` *if a conflict is detected.*

– $\texttt{check}_{\mathcal{T}}\colon \mathsf{Mealy} \to (\Sigma^\star + \mathbb{1})$ *performs a consistency check of a given Mealy machine hypothesis against the observation tree* $\mathcal{T}$*, at no system interaction cost. It optionally returns a counterexample if found.*

– $\texttt{test}_{\mathcal{T}}\colon \mathsf{Mealy} \to (\Sigma^\star + \{\texttt{Prune}\})$ *is the function used to test the system for counterexamples. It takes a hypothesis proposed by the learner and coherent with* $\mathcal{T}$*, and tests the system until a counterexample or a conflict is found. The tests are taken from* `sampleWord` *which is instantiated with off-the-shelf test suite algorithms, such as the Wp-Method [Fuj+91] or Hybrid-ADS [LY94; Sme+15].*

△

Subroutine 5.1: $\texttt{read}_{\mathcal{T}}$

Data: Input word $i \in \Sigma^\star$.

Result: output word $o \in \Gamma^\star$ or `Prune`.

1 $o \leftarrow [\![\mathcal{T}]\!](i)$; // attempt observation tree first

2 **if** $o \in \Gamma^\star$ **then return** o ;

3 $o \leftarrow sys(i)$; // fallback to observing the system

4 **if** $\texttt{write}_{\mathcal{T}}(i,o) = \texttt{Prune}$ **then return** `Prune` ;

5 **return** o;

Subroutine 5.2: $\texttt{write}_{\mathcal{T}}$

Data: Observation $(i,o) \in \Sigma^\star \times \Gamma^\star$.

Result: Updated $\mathcal{T}$, optional `Prune`.

```
1  conflict ← ⊥;
2  ⟨Q, q_ε, Σ, Γ, δ, λ⟩ ← T;
3  q_w ← q_ε;
4  for (c_i, c_o) ∈ zip(i, o) do                    // iterate over each character pair
5      if λ(q_w, c_i) = c'_o ∈ Γ ∧ c'_o ≠ c_o then
6          conflict ← ⊤; delete(q_{w·c_i});          // remove q_{w·c_i} and all its successors
7      Q ← Q ∪ {q_{w·c_i}};                          // update the tree's structure if needed
8      δ(q_w, c_i) ← q_{w·c_i}; λ(q_w, c_i) ← c_o;
9      q_w ← q_{w·c_i};
10 if conflict then return Prune ;
```

Subroutine 5.3: $\texttt{check}_{\mathcal{T}}$

Data: Hypothesis $\mathcal{H} \in$ Mealy.

Result: Optional cex $i \in \Sigma^\star$.

```
1  for (i, o) ∈ L_T do
2      if [[H]](i) ≠ o then return i ;
```

Subroutine 5.4: $\texttt{test}_{\mathcal{T}}$

Data: Hypothesis $\mathcal{H}$ consistent with $\mathcal{T}$.

Result: cex $i \in \Sigma^\star$ or `Prune`.

```
1  while ⊤ do
2      i ← sampleWord(); o ← sys(i);
3      if write_T(i, o) = Prune then
4          return Prune
5      if [[H]](i) ≠ o then
6          return i
```

While all four functions of the Reviser rely on the observation tree, the $\texttt{write}_{\mathcal{T}}$ function is the only one that actually changes the tree itself, and is responsible for reconciliation of conflicts as they arise. By causality of Mealy machines (see Section 2.2.1), a mismatch between a new observation and the tree at any position

invalidates every observation extending that position, so the reconciliation amounts to discarding the affected subtree and applying the new observation.

Example 5.2.8. *In Example 5.2.4 the right-hand tree is the result of updating the observation tree with observation* (aaa, abb) *via* $\texttt{write}_{\mathcal{T}}(aaa, abb)$*, resulting in the conflicting observations* (aaa, aab) *and* (aab, aaa) *being removed.*

We now characterize the language maintained by the Reviser: Given a stream of observations, which observations can be retrieved from the tree at any given step? The invariant below answers this, capturing our most-recent-update policy precisely. The argument proceeds in two stages: first, we characterize the effect of a single call to $\texttt{write}_{\mathcal{T}}$, and then we iterate this characterization over the stream.

We begin by analyzing a single application of $\texttt{write}_{\mathcal{T}}(i,o)$. The algorithm iterates through the character pairs $(c_i, c_o) \in \mathsf{zip}(i,o)$, walking the tree along the input. At each iteration, it examines $\lambda(q_w, c_i)$, where q_w is the state corresponding to the input word consumed so far. Three cases arise:

(Match) $\lambda(q_w, c_i) = c_o$: the tree already records the expected transition, so no modification is made.

(Extend) $\lambda(q_w, c_i) = \bullet$: the tree has no transition at (q_w, c_i), so a transition to a fresh state $q_{w \cdot c_i}$ is added with $\lambda(q_w, c_i) \leftarrow c_o$.

(Conflict) $\lambda(q_w, c_i) = c_o' \in \Gamma$ with $c_o' \neq c_o$: the tree records a conflicting transition, so the subtree rooted at $q_{w \cdot c_i}$ is deleted, a fresh state $q_{w \cdot c_i}$ is attached in its place, and $\lambda(q_w, c_i) \leftarrow c_o$.

Both (Extend) and (Conflict) leave $q_{w \cdot c_i}$ as a fresh leaf with no outgoing transitions, so all subsequent iterations must be (Extend). Consequently, considering the iterations of $(c_i, c_o) \in \mathsf{zip}(i,o)$, the execution path is composed strictly of a (Match)-prefix and an (Extend)-suffix, separated by at most one (Conflict) iteration.

Lemma 5.2.9 (Step invariant of `write`)**.** *Let* $\mathcal{T}$ *be a tree-shaped observation tree and* $(i,o) \in \Sigma^\star \times \Gamma^\star$ *an observation. Let* $\mathcal{T}'$ *denote the tree resulting from* $\texttt{write}_{\mathcal{T}}(i,o)$.

Then $\mathcal{T}'$ is tree-shaped with:

$$\mathcal{L}_{\mathcal{T}'} = \big(\mathcal{L}_{\mathcal{T}} \setminus C_{(i,o)}\big) \cup \mathsf{prefixes}(i,o) \quad \textit{where} \quad C_{(i,o)} \triangleq \{(i',o') \mid (i',o') \not\downarrow (i,o)\}$$

Proof. The tree-shape of $\mathcal{T}'$ is preserved by each case: (Match) does not change the tree, (Extend) adds a leaf state, (Conflict) removes a subtree and attaches a leaf state. For the tree's language we case-split on the shape of the execution.

All (Match). Then $\mathcal{T}' = \mathcal{T}$, and every transition traversed by (i,o) agrees with the tree; hence $\mathsf{prefixes}(i,o) \subseteq \mathcal{L}_{\mathcal{T}}$. No observation in $\mathcal{L}_{\mathcal{T}}$ conflicts with (i,o): any witness would require a common input prefix of i and the observation's input on which outputs disagree, contradicting (Match) along the entirety of i. Hence $\mathcal{L}_{\mathcal{T}'} = \mathcal{L}_{\mathcal{T}} = (\mathcal{L}_{\mathcal{T}} \setminus C_{(i,o)}) \cup \mathsf{prefixes}(i,o)$.

(Match)-prefix followed by (Extend)-suffix, no (Conflict). Let u be the prefix consumed by the (Match)-iterations, so the first (Extend) occurs at prefix u with characters (c_i, c_o). The (Match)-iterations leave $\mathcal{T}$ unchanged, so at the pivot iteration $\lambda(q_u, c_i) = \bullet$ in $\mathcal{T}$. By tree-shape, no observation in $\mathcal{L}_{\mathcal{T}}$ extends $u \cdot c_i$. Hence no conflict witness involving a common prefix longer than u exists, and by the (Match)-agreement no witness on prefixes of length $\leq |u|$ exists either. Thus $C_{(i,o)} \cap \mathcal{L}_{\mathcal{T}} = \varnothing$. The (Extend)-suffix then adds exactly $\mathsf{prefixes}(i,o) \setminus \mathsf{prefixes}(u, o[0..|u|-1])$ to the language without removing anything; combined with the (Match)-prefix's observation that $\mathsf{prefixes}(u, o[0..|u|-1]) \subseteq \mathcal{L}_{\mathcal{T}}$, we obtain $\mathcal{L}_{\mathcal{T}'} = \mathcal{L}_{\mathcal{T}} \cup \mathsf{prefixes}(i,o) = (\mathcal{L}_{\mathcal{T}} \setminus C_{(i,o)}) \cup \mathsf{prefixes}(i,o)$.

(Match)-prefix, (Conflict), (Extend)-suffix. Let u be the prefix consumed by the (Match)-iterations, so the (Conflict) occurs at prefix u with character pair (c_i, c_o). By the (Match)-prefix argument, $\mathcal{T}$ is unchanged entering the pivot iteration, and $\lambda(q_u, c_i) = c_o' \neq c_o$ in $\mathcal{T}$. The (Conflict) deletes the subtree rooted at the existing $q_{u \cdot c_i}$. By tree-shape, the observations removed from $\mathcal{L}_{\mathcal{T}}$ are exactly those whose inputs extend $u \cdot c_i$. Each such observation (i', o') has $u \cdot c_i \in \mathsf{prefixes}(i') \cap \mathsf{prefixes}(i)$ and $o'[|u|] = c_o' \neq c_o = o[|u|]$, so $u \cdot c_i$ is a conflict witness. Conversely, every element of $C_{(i,o)} \cap \mathcal{L}_{\mathcal{T}}$ has such an input prefix, as any earlier witness is ruled out by the (Match)-agreement on prefixes of u. Hence the removed observations are exactly

$C_{(i,o)} \cap \mathcal{L}_{\mathcal{T}}$. The (Extend)-suffix then adds the remaining prefixes of (i,o), giving $\mathcal{L}_{\mathcal{T}'} = (\mathcal{L}_{\mathcal{T}} \setminus C_{(i,o)}) \cup \mathsf{prefixes}(i,o)$. □

The following lemma characterizes the information represented by the Reviser's observation tree at any point in the learning process. Its language contains the maximal amount of non-conflicting information observed so far, with more recent observations taking priority whenever conflicts arise.

Lemma 5.2.10 (Reviser Language)**.** *Let $\mathcal{T}_0$ be the empty observation tree, and let $(i_k, o_k)_{k\geq 1}$ be a stream of observations applied successively, where $\mathcal{T}_k$ is the tree resulting from $\mathtt{write}_{\mathcal{T}_{k-1}}(i_k, o_k)$. For every $K \geq 0$:*

$$\mathcal{L}_{\mathcal{T}_K} = \{(\varepsilon,\varepsilon)\} \cup \Big\{ (i,o) \;\Big|\; \exists k.\ 1 \leq k \leq K \wedge (i,o) \in \mathsf{prefixes}(i_k,o_k) \\ \wedge \nexists k'.\ k < k' \leq K \wedge (i,o) \not\downarrow (i_{k'},o_{k'}) \Big\}$$

Proof. By induction on K. Let $R(K)$ denote the right-hand side.

Base case ($K = 0$). The empty tree has $\mathcal{L}_{\mathcal{T}_0} = \{(\varepsilon,\varepsilon)\}$, and $R(0) = \{(\varepsilon,\varepsilon)\}$ as the set comprehension is vacuous.

Inductive step. Assume $\mathcal{L}_{\mathcal{T}_K} = R(K)$, and that $\mathcal{T}_K$ is tree-shaped (by Lemma 5.2.9 and induction). Let $C_{K+1} \triangleq \{(i,o) \mid (i,o) \not\downarrow (i_{K+1},o_{K+1})\}$, and let $S(K)$ denote the set comprehension in $R(K)$, so $R(K) = \{(\varepsilon,\varepsilon)\} \cup S(K)$.

We first show $S(K+1) = (S(K) \setminus C_{K+1}) \cup \mathsf{prefixes}(i_{K+1},o_{K+1})$. Splitting $S(K+1)$ by whether the witness index k satisfies $k \leq K$ or $k = K+1$: the $k \leq K$ branch is $S(K)$ further restricted to exclude elements conflicting with (i_{K+1},o_{K+1}), i.e. $S(K) \setminus C_{K+1}$; the $k = K+1$ branch has empty k'-range and reduces to $\mathsf{prefixes}(i_{K+1},o_{K+1})$. The two branches may overlap, but their union gives the claimed equality.

By definition of $\not\downarrow$, $(\varepsilon,\varepsilon) \notin C_{K+1}$ (the only candidate common input prefix is ε,

on which output comparison is vacuous), and $(\varepsilon,\varepsilon) \in \mathsf{prefixes}(i_{K+1}, o_{K+1})$. Hence:

$$\begin{aligned} R(K+1) &= \{(\varepsilon,\varepsilon)\} \cup S(K+1) \\ &= \{(\varepsilon,\varepsilon)\} \cup (S(K) \setminus C_{K+1}) \cup \mathsf{prefixes}(i_{K+1}, o_{K+1}) \\ &= (\{(\varepsilon,\varepsilon)\} \cup S(K)) \setminus C_{K+1} \cup \mathsf{prefixes}(i_{K+1}, o_{K+1}) \qquad ((\varepsilon,\varepsilon) \notin C_{K+1}) \\ &= (R(K) \setminus C_{K+1}) \cup \mathsf{prefixes}(i_{K+1}, o_{K+1}) \end{aligned}$$

By the induction hypothesis $R(K) = \mathcal{L}_{\mathcal{T}_K}$, and by Lemma 5.2.9 applied to $\mathcal{T}_K$:

$$\mathcal{L}_{\mathcal{T}_{K+1}} = (\mathcal{L}_{\mathcal{T}_K} \setminus C_{K+1}) \cup \mathsf{prefixes}(i_{K+1}, o_{K+1}) = R(K+1) \qquad \square$$

5.2.3 Interface Implementation

With the Reviser's functions and implemenations established, we can now build the interface described in Sec. 5.2.1. This mostly amounts to implementing membership and equivalence queries, as the testing interface is simply composed of calls to *sys*.

Membership queries can be implemented, for $i \in \Sigma^\star$, as $mq(i) \triangleq \mathtt{read}_{\mathcal{T}}(i)$. In the case that a `Prune` signal is returned instead, this is intercepted for MAT learners, and handled directly. When $\mathcal{T}$ does not have the answer to this particular query, $\mathtt{read}_{\mathcal{T}}$ sends it through to the SUL (with the call to *sys*) and the result is applied in $\mathcal{T}$. This process is illustrated in Figure 5.3.

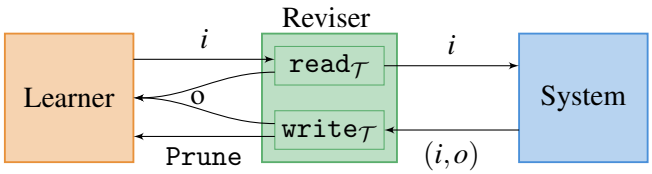


Figure 5.3: Implementation of Membership Queries (*mq*) in the Reviser.

Equivalence queries are handled in two steps. First, the hypothesis given by the learner is checked against the observation tree using $\mathtt{check}_{\mathcal{T}}$. If a counterexample is found, it is returned. Otherwise, the Reviser tests the System to discover new information and update the tree. If a counterexample is found, it is returned to the learner, or if a conflict arose then the learner is pruned. (Figure 5.4).

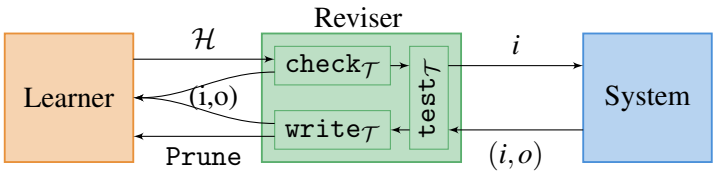


Figure 5.4: Implementation of Equivalence Queries (*eq*) in the Reviser.

Remark 5.2.11 (Modularity)**.** *We present* CƐAL *in the case of* closed-box *learning where we do not have access to any information from the system but can interact with it. However, the framework is fully modular: one can interface any model-checking (or other) method before the calls to system in* $\texttt{test}_\mathcal{T}$ *and* $\texttt{read}_\mathcal{T}$ *when related models (specifications, parts of the System, etc.) are available, allowing* CƐAL *to perform open-box learning.* CƐAL *focuses on the* storage *and maintenance of information, rather than on its acquisition.*

5.2.4 Learning Lifecycle

So far we have specified the Reviser operationally, function by function. We now consider what an execution of CƐAL looks like as a whole. A run is driven entirely by the Learner, as the Reviser performs work only in response to *mq* and *eq* calls, and the System is touched only when the observation tree $\mathcal{T}$ cannot answer on its own. A conflict is handled in the same reactive fashion: the Reviser returns `Prune`, and the Learner restarts. Because $\mathcal{T}$ retains every non-conflicting observation, the restart requires no further tests on the System until the Learner reaches fresh territory.

This shape leaves three pragmatic choices that a MAT description elides. In the rest of this section we consider which hypothesis to return as the final model, when to halt, and how to exploit $\mathcal{T}$ beyond the minimal interface.

Hypothesis Selection Active automata learning involves the production of a sequence of hypotheses that are refined over time, with the goal of converging towards a correct one. As such, a key characteristic of different approaches to automata learning is how a final model is to be selected, out of the many hypotheses. In the case of MAT this is simple: Learning produces a sequence of ever more accurate models, until termination occurs with a positive equivalence query. It is then logical

to pick the most recently produced hypothesis as the final model. However, when dealing with conflicts, this is no longer necessarily the case for CℰAL. In particular, when electing a model out of a sequence of hypotheses, CℰAL provides two options:

– *Most Recent*: This hypothesis selection strategy is the one known classically: the most recently produced hypothesis is the one to be elected as final. This strategy is sensible in the case of learning with no noise, or in the case of learning targets that evolve over time (see Section 5.1.2). In this case each hypothesis is prefered to its predecessor, as it either refines the hidden Myhill–Nerode congruence further, or captures behavior pertaining to the newer target.

– *Most Frequent*: In this selection strategy, the sequence of hypotheses is analyzed to elect a final model. We count the frequency of each unique model (up to language equivalence) over the sequence, and elect the most frequently occurring one. This strategy makes sense when dealing with noise (see Section 5.1.1), as we may be producing (rarely) hypotheses that capture noisy behavior that is fixed over time. As such, we want to select not the latest model produced, but the one that is the most stable. This strategy can be implemented efficiently in practice (using hash fingerprints and counters, for example) and on-the-fly during learning, allowing us to not have to store the whole sequence of hypotheses as it is produced.

Termination Classical MAT learners rely on a conflict-free environment to make monotonic progress towards the target, and their termination is guaranteed, in theory, by the teacher's equivalence oracle that can certify when a hypothesis is correct. In practice, such an oracle is unrealizable: a finite Mealy machine admits a finite complete test suite, but in a closed-box setting we have no bound on its size, and therefore no way to know when enough tests have been run. Termination is instead achieved via an *explicit* success metric. This is typically the hypothesis surviving a predefined test suite, or an observation budget being exhausted. This pragmatic notion of termination is usually left implicit in MAT, folded into the equivalence oracle, while CℰAL makes it explicit. Because the Reviser drives the testing of the system (via $\texttt{test}_{\mathcal{T}}$), it is also the natural place to enforce such a criterion: the Reviser

halts the learning process according to the termination criterion chosen, returning the currently selected hypothesis. In our experimental evaluation (Section 5.3), we terminate when the currently selected hypothesis has survived a complete test suite, or when a predefined observation budget is reached.

Native Optimizations The separation between the Learner and the Reviser in CƐAL enables two optimizations over a straightforward MAT deployment, arising from the fact that the observation tree is already a first-class record of the observations made.

– *Query Caching*: Classical MAT learners typically maintain their own membership-query cache to avoid re-observing the system. In CƐAL, the observation tree already plays this role (as made explicit by the definition $mq(i) \triangleq \texttt{read}_{\mathcal{T}}(i)$), so the Learner can forgo a separate cache entirely. This is particularly natural for learners whose internal state is itself an observation tree, such as $L^\sharp$ [Vaa+22] or the observation-tree-based learner of Soucha and Bogdanov [SB20]: for these one does not need to maintain a separate Learner-side data structure, and the Learner can produce hypotheses, and reason directly from the observation tree $\mathcal{T}$.

– *Specialized Pruning*: To support the minimal MAT learner interface, CƐAL handles the `Prune` signal by restarting the Learner. This restart incurs no additional observation cost, since the Reviser replays all prior observations from $\mathcal{T}$ on demand (see Section 5.2.1). For specific Learner data structures, this coarse restart can be refined to discard only the portion affected by the conflict, a refinement that is necessarily algorithm-specific. A CƐAL-aware Learner can read $\mathcal{T}$ directly and reconstruct the unaffected portion of its own data structure from it, as the IKV algorithm of Chapter 4 does for the classification tree.

5.3 Experimental Evaluation

We introduced CƐAL to extend the power of classic MAT learners to environments where observations may conflict. Recent work in adaptive automata learning, such as the IKV algorithm of Chapter 4, made progress on the specific case of learning with conflicts arising from systems that evolve at discrete, known times. These methods are, however, not suitable for other sources of conflict, such as observation *noise*.

In practice, each symbol sent to or received from the SUL can be corrupted by noise, making longer observations more likely to be affected overall. Recall that a poor choice for the number of observation samples (i.e., the repetitions of each query) can lead to a learning failure (if noisy observations reach the learner uncorrected), or a timeout (if too many repetitions are made). Hence, noise affects not only the *efficiency*, but also the *success rate* of the learning, i.e., the proportion of runs that return a correct model.

In this section we present a wide range of experiments conducted to determine how well CƐAL can mitigate such negative effects. We first discuss the implementation of CƐAL, followed by our research questions and the methodology of our evaluation. We then present the results from said experiments, and analyze this data to answer our research questions.

Implementation We have implemented the CƐAL framework in approximately 2,500 lines of Java, as part of the LearnLib library [FHS25]. LearnLib provides implementations of many existing classic MAT algorithms, such as $L^\star$, KV, and TTT. Any existing LearnLib active learning algorithm can automatically be used in CƐAL. We have additionally contributed an implementation of $L^\sharp$, which at the time of publication was absent from LearnLib. This implementation has since been vetted and adopted by the official LearnLib distribution. In our experiments we will make use of all four of these MAT learners for performance comparisons.

5.3.1 Research Questions and Methodology

We are interested in the following research questions (RQs) on the performance of CƐAL compared to state-of-the-art MAT approaches:

RQ1 Across different realistic targets, types of noise, and noise levels, does CƐAL achieve higher success rates than MAT?

RQ2 Across the same conditions, does CƐAL use fewer observations than MAT?

To answer these research questions, we have designed an experiment set covering multiple realistic targets, several automata learning algorithms working over

both MAT and CℰAL, and progressive levels of noise and their mitigations. We now detail and motivate the design decisions in these experiments.

Benchmarks Ideally, a new learning approach should be evaluated on real targets. Since this is rarely feasible at scale, we instead use 36 Mealy machines that model real systems from previous successful applications of automata learning [Nei+19]. These targets range in size from 4 to 66 states, with alphabets of 7 to 22 input symbols. Notably, some of these systems exhibited real input and output noise, and their reference models were obtained by MAT learning with high sampling rates to filter noise out [FJV16; Fit+17]. This makes them particularly suitable as benchmarks for noise-robust learning.

Observation Noise We simulate different types and levels of observation noise by intercepting interactions with the SUL and modifying the symbols on-the-fly. Specifically, we consider either *input* or *output* noise (see Section 5.1.1) occurring with noise rates η of 0.01, 0.05 and 0.1. This means that for each given symbol of an observation, there is a probability η of the symbol being artificially flipped to a different, uniformly random symbol of the same alphabet set.

Query Sampling To mitigate the presence of noise in the observations, and as commonly done in practice (see Section 5.1.1), we employ *query sampling*, where each requested membership query from the learner is sampled, and majority-voted in an attempt to lower the effects of noise. Each observation is repeated at least *min* times, and at most *max* times, while aiming to achieve an outcome with 80% majority. We conduct our experiments over three increasing levels (min, max) of sampling: $(5, 10)$, $(10, 20)$, and $(20, 30)$.

Learning Algorithms We instantiate both MAT and CℰAL with the $L^\star$, KV, TTT, and $L^\sharp$ algorithms of LearnLib. Notably, $L^\star$ is implemented with the improvements of Rivest and Schapire [RS93] for counterexample handling. These cover the most widely used, performant active automata learning algorithms.

Fixed Settings Beyond the parameters above, all experiments share the following fixed settings. An artificial cache layer is used for MAT's observations, and the native cache is used for CℰAL. When we report the number of observations (or their

symbols), we mean strictly *true* observations on the SUL, i.e., from cache misses. For CℰAL, we use the *Most Frequent* hypothesis selection strategy for dealing with conflicts from noise, as motivated in Section 5.2.4. For MAT equivalence queries, and for implementing CℰAL's `sampleWord` method, we use the Hybrid-ADS testing algorithm [LY94]; preliminary experiments supported this choice over the Wp-Method [Fuj+91] and random testing for both frameworks.

Metrics and Reproducibility Each learning run is bounded by a budget of 10 million observations, after which it is marked as a failure. For each unique combination of parameters we perform 100 learning runs with varying random seeds, and average the results to reduce the variance introduced by the multiple sources of randomness in the learning process. We have found this number to be adequate in practice, while keeping the total number of learning runs experimentally feasible given the over 18,000 configurations in our cross-product of parameters. By *success rate* of an experiment we mean the percentage of said runs that terminated with a minimal, equivalent model of the target.

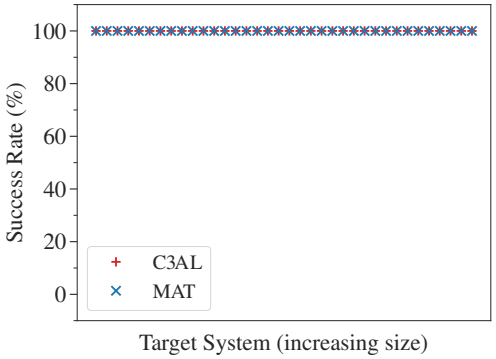
5.3.2 Results and Analysis

Our experimental setup has produced over 18,000 distinct experiments, across close to 2 million learning runs. While presenting the totality of this data is naturally not feasible, we consider, for each combination of independent variables, which framework (under which algorithm) achieves the most successful and, within that, most efficient configuration.

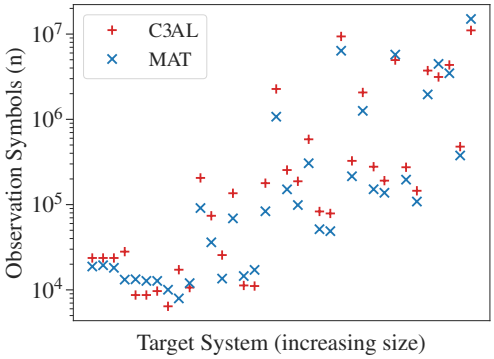
In the graphs below (Figures 5.5 to 5.10) we plot, for each target, noise type, and noise rate, the success rate and number of symbols used in observations, where each framework is allowed to choose its own best parameter profile. We additionally provide all the data points used in Appendix B.

We now analyze the results of these experiments to draw high-level conclusions about how MAT and CℰAL compare, and to answer our research questions.

Success Rates First and foremost, we discuss the impact of different parameters on the success rates of the experiments. We can see from the graphs that, as expected, while the exact type of noise does not have a significant impact on success rates,

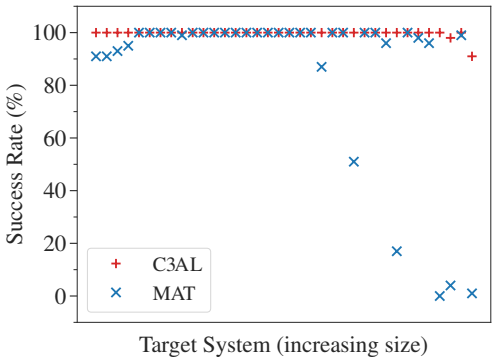


(a) Success rate (higher is better)

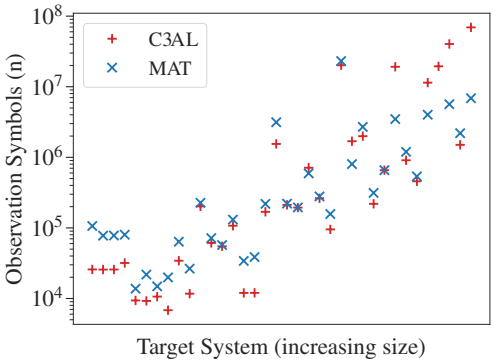


(b) Average symbol use (lower is better)

Figure 5.5: INPUT noise at **0.01**.

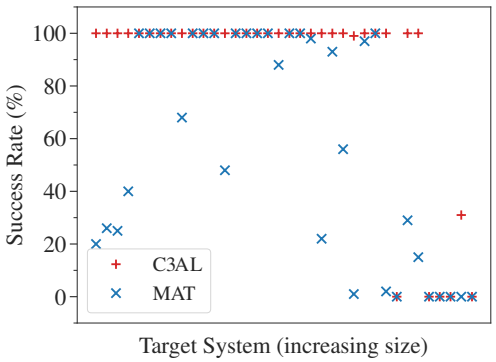


(a) Success rate (higher is better)

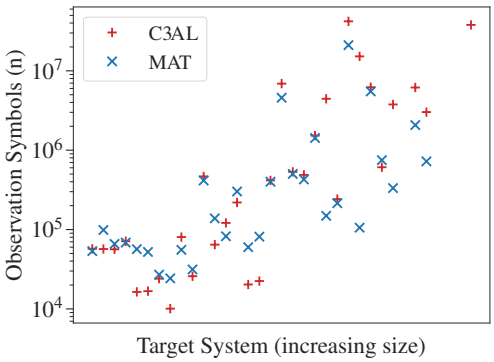


(b) Average symbol use (lower is better)

Figure 5.6: INPUT noise at **0.05**.

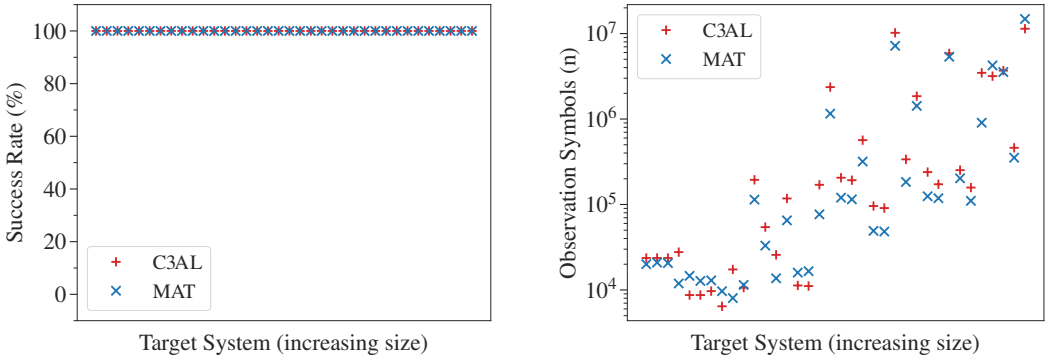


(a) Success rate (higher is better)



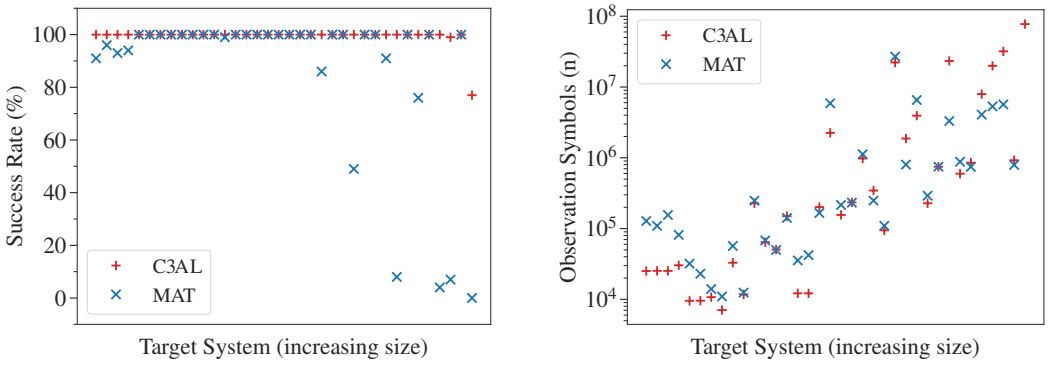
(b) Average symbol use (lower is better)

Figure 5.7: INPUT noise at **0.1**.



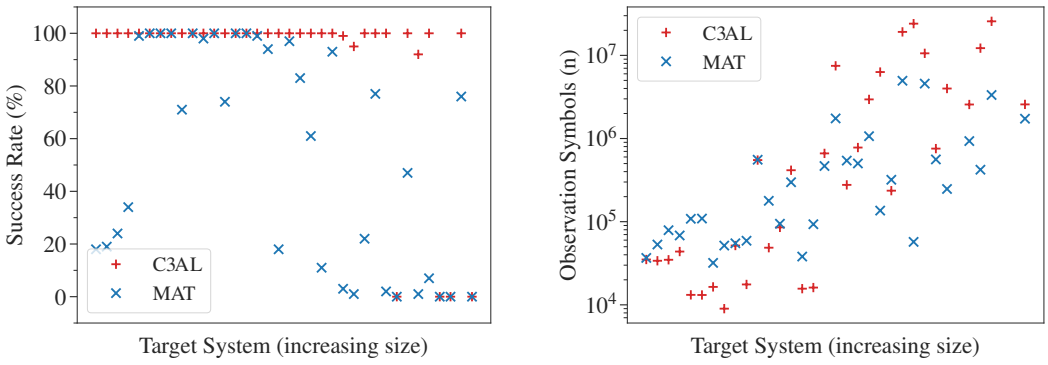
(a) Success rate (higher is better) **(b)** Average symbol use (lower is better)

Figure 5.8: OUTPUT noise at **0.01**.



(a) Success rate (higher is better) **(b)** Average symbol use (lower is better)

Figure 5.9: OUTPUT noise at **0.05**.



(a) Success rate (higher is better) **(b)** Average symbol use (lower is better)

Figure 5.10: OUTPUT noise at **0.1**.

the *level of noise* does. In particular, at a very low noise rate of 0.01 (Figures 5.5a and 5.8a) both frameworks are capable of maintaining perfect success rates. However, once the rate increases to 0.05 (Figures 5.6a and 5.9a), MAT's success rate starts to fluctuate, with extremely low success rates for bigger targets. CƐAL is slightly affected by an increase in noise, but overall maintains a success rate close to 100%. Once the noise is increased to its highest level, 0.1 (Figures 5.7a and 5.10a), we can see that MAT's success rates drop significantly overall, while CƐAL's remain high across most targets, decreasing only for the largest ones at this level of noise. Even there, CƐAL remains the more reliable of the two frameworks.

Efficiency The observation symbol count graphs (Figure 5.5b to 5.10b) show an expected picture: larger systems require more interactions to be learned. Importantly, while MAT appears efficient on large noisy targets, its success rates there are considerably lower, and efficiency is only meaningful conditional on a correct hypothesis. The low symbol counts for MAT in these regimes are not the typical cost of a successful run; they reflect the rare runs in which noise happened to be sparse enough to be filtered cheaply, which is also what allowed the run to succeed in the first place. Restricting the comparison to targets where both frameworks have high success rates, MAT is slightly more efficient than CƐAL at a low noise rate of 0.01, but CƐAL uses fewer symbols as soon as the noise rate is increased.

Overall Results Perhaps most importantly, CƐAL provides the most efficient *correct* configuration in 70% of the experiments, either with a higher success rate than MAT, or with the same success rate at a lower average observation count. We provide this result for each individual experiment in Appendix B. In particular, in every experiment CƐAL achieves a success rate that is at least as high as MAT's, often outperforming it. In addition to this, experiments run with CƐAL had an overall success rate of 95.5% compared to MAT's 79.5% success rate. This alone has allowed CƐAL to perform successful runs that no configuration of MAT was able to perform, namely learning moderate to large targets at 0.1 noise rate.

We found that much of CƐAL's advantage on higher noise rates stems from its ability to succeed with a *lower number of repeats* than MAT requires. This supports

the intuition that reducing the number of query samples has a positive impact on performance. Taken together, these results answer RQ1 and RQ2 in the affirmative. On success rates, CƐAL is never worse than MAT and is substantially better at higher noise. On efficiency, the picture is regime-dependent: at a low noise rate of 0.01, MAT is very slightly more efficient than CƐAL; elsewhere, on runs where both frameworks succeed, CƐAL is often the more efficient of the two. We therefore conclude that, for noisy environments, CƐAL provides a better learning environment than state-of-the-art MAT approaches, particularly as noise increases.

5.3.2.1 Other Findings

We now report on other findings not directly needed for answering our research questions, but interesting nonetheless.

Alphabet Impact Firstly, target size is not a sole determining factor in learning difficulty. For example, in the success graphs for rates 0.05 and 0.1 (Figures 5.6a, 5.7a, 5.9a and 5.10a), we can see that the four smallest targets prove to be harder to learn with noise than the next seventeen, despite being smaller. This is because these smaller targets have comparatively large alphabets; indeed, their alphabets are twice the size of the next well-performing targets. This is expected, as a larger alphabet increases the number of observations required per state, and with it the probability that a given observation is corrupted.

Testing Relevance Secondly, as already accepted by the community [Vaa17; ATW24], we can confirm that indeed most of the observation spent in learning are used to realize equivalence queries. In particular, we found that equivalence tests account for 89.1% of observations in MAT, 59.8% in CƐAL, and 66.3% on average. This reinforces the need for tighter integration between testing and learning, and for testing algorithms tailored to the specific demands of automata learning [KJR24], where the dominant cost lies not in individual membership queries but in the equivalence procedures that drive hypothesis refinement.

Algorithm Performance Finally, and perhaps most surprisingly, the commonly accepted ordering of learner efficiencies did not surface in our experiments. From expectedly most performant to least we have $L^\sharp$, TTT, KV, and $L^\star$, based on

complexity analyzes and classic benchmarks in MAT. Through our experiments we have found that, at least in our particular case of closed-box learning of Mealy machines with noise and randomized testing algorithms, this ordering cannot be seen in the results. The best configurations for each framework were not consistent on which algorithm performed the best. Not only was there no clear "winning" algorithm, we found no pattern based on noise, target and alphabet size, or number of repeats that had a relevant correlation to the better performance of any one algorithm.

We believe that this is not a flaw of the complexity analyzes or benchmarks themselves, but rather highlights the need to evaluate learning algorithms under the realistic conditions in which they are actually deployed: with noise, cache layers, query sampling and repetitions, and equivalence testing algorithms. Complexity analyzes in MAT necessarily abstract these factors away, and classic MAT benchmarks do not expose algorithms to the perturbations that dominate practical runs. As a consequence, small theoretical advantages between algorithms are readily overshadowed by the variance introduced by real-world factors. Only by studying learners under conditions representative of real use can we obtain a faithful understanding of their performance in practice.

5.4 Related Work

There has been extensive work on applying classical learning algorithms like $L^\star$ [Ang87], KV [KV94], TTT [IHS14], and $L^\sharp$ [Vaa+22] to real-world systems such as passports [ASV10], network protocol implementations [FJV16; Fit+17; Fit+20], and bank cards [ARP13]. These works typically handle noise through *ad hoc* techniques that are neither studied within MAT nor reusable across learners. By replacing the teacher with the SUL and the Reviser, CƐAL addresses noise at the framework level, independently of the choice of learner.

Closest to our infrastructure, the LearnLib library [FHS25] provides *caches* that can be placed in the learning environment to avoid repeated queries, playing a role similar to that of CƐAL's use of observation trees. The Reviser, however, goes further: it acts as both membership and equivalence oracle, tests the system directly,

and resolves conflicts by pruning the learner's data structure in a query-efficient and correctness-preserving manner.

A separate line of work improves the efficiency of learning under mutating targets by reusing previously learned behavior. *Adaptive* learning algorithms [Cha+08; DMD19; HMV18; Win+13; KJR25], including the IKV algorithm of Chapter 4, start from pre-seeded information from previous runs and either confirm or filter out knowledge that no longer applies. CƐAL builds on this direction, but autonomously corrects itself when faced with conflicts, continuously monitoring the system for changes without requiring any notification of mutations. This allows it to be applied to fully closed-box systems, unlike IKV and the adaptive learner of Kruger et al. [KJR25], both of which require explicit notification of mutations.

Finally, Khmelnitsky et al. [Khm+22] present an empirical analysis of the robustness of the $L^\star$ algorithm under noise, without relying on query sampling. They study several noise models, including the random input and output noise we consider here. Their setting differs from ours in an important respect: the noise they study is deterministic, producing a perturbed but *fixed* target language against which $L^\star$ learns without ever encountering conflicting observations. Their analysis therefore focuses on how much the learned model diverges from the noise-free language, and how this divergence evolves over learning rounds. Our setting instead considers probabilistic noise that produces conflicts during learning, which is the phenomenon CƐAL is designed to handle. We note, nonetheless, that their results support one of our design choices: they show empirically that the proximity of the learned hypothesis to the noise-free target language does not grow monotonically across learning rounds, reinforcing our decision to forgo the practice of selecting the most recent hypothesis, and instead adopt the *Most Frequent* selection strategy in CƐAL.

5.5 Discussion

In this chapter we introduced CƐAL, a framework for active automata learning under conflicting observations. CƐAL departs from MAT by making the SUL a first-class component of the framework, and introducing the *Reviser*: an agent responsible for

testing the system, curating observations, and maintaining a conflict-free view for the learner. This separation allows classical MAT algorithms to be reused unchanged within a framework that is robust to both observation noise and system evolution.

We evaluated CƐAL on a large body of experiments over 36 realistic targets, multiple learning algorithms, and a range of noise types and levels, totaling over 18,000 configurations and close to 2 million learning runs. CƐAL achieved an overall success rate of 95.5% against MAT's 79.5%, and in 70% of experiments provided the most efficient correct configuration. Most importantly, CƐAL enabled the successful learning of targets that no MAT configuration could handle, typically complex systems under high noise.

Several directions for future work follow naturally from these results. First, given the dominance of equivalence testing in the overall cost of learning, we see strong motivation for the design of testing algorithms tailored to the specific demands of active learning, rather than repurposing general-purpose conformance testing methods. Second, CƐAL's modular architecture lends itself naturally to *open-box* extensions, where the Reviser draws information from additional sources such as specifications or source code access. This would offset the cost of equivalence queries by exploiting cheaper sources of observation when searching for counterexamples. Finally, an evaluation of CƐAL on a real case of system evolution would both validate the framework in deployment and provide an opportunity to fine-tune it for that specific use case.

Chapter 6

SMT-Based Active Learning of Weighted Automata

So far, this thesis has concerned itself with automata and transducers in the Boolean domain: a regular language either contains a word or not, and a Mealy machine either produces a given output word or it does not. Many systems of practical interest, however, have more nuanced semantics that do not fall cleanly into this setting. We may instead want to reason about the *cost* of an execution, the *probability* of accepting an input, or the *resources* it consumes.

Weighted finite automata (WFAs) generalize classical automata to capture exactly this kind of quantitative behavior. Transitions carry weights drawn from a semiring, allowing WFAs to model relevant quantities, and then serve as a basis to answer a range of important questions in quantitative verification and performance analysis: *What is the minimal cost of reaching an error state?*, *Is the probability of failure below a threshold?*, *Does every execution respect a given resource bound?*. The question whether $L^\star$-style algorithms can be extended to WFAs was a natural one to tackle, and the last decade saw a range of theoretical results, mostly showing termination conditions for different classes of semirings: starting from fields [BV96] to principal ideal domains [Van+20] or number rings [Ari+25]. Maybe not surprisingly, learning WFAs turned out to be substantially more challenging than learning deterministic finite automata. Classical Hankel- and table-based generalizations of $L^\star$ must reason about algebraic structure in the weight domain (the semiring), and

proving termination turned out to be very subtle over many natural semirings, often relying on indirect linear-algebraic arguments. In practice, these challenges limit both the applicability and the scalability of existing WFA learning algorithms.

This chapter takes a different, algorithmically-driven approach to learning weighted automata. Rather than refining a Hankel matrix or observation table, we cast learning directly as a constraint-solving problem. Building on a coalgebraic view of WFAs, we derive a system of equations over the given semiring whose solutions correspond exactly to nondeterministic WFAs that agree with the teacher on all observed queries. Learning then amounts to incrementally strengthening this equation system in response to counterexamples and invoking an SMT solver to search for a solution. This perspective yields an algorithm that is parametric in the semiring, naturally supports nondeterminism, and — when it terminates — produces minimal weighted automata by construction.

A key advantage of this formulation is its compatibility with modern SMT technology. We show that the resulting equation systems fall into decidable arithmetic theories for several practically relevant semirings, including all finite semirings and infinite ones such as the tropical semiring. We further identify sufficient conditions for termination, which in particular ensure termination for all finite semirings. An extensive experimental evaluation demonstrates that our SMT-based learner can infer numerous minimal WFAs over both finite and infinite semirings, outperforming a naive baseline and competing favorably with an existing algorithm while producing significantly smaller automata and requiring less teacher interaction. Taken together, our results position constraint-based learning as a practical and robust alternative to Hankel/$L^\star$-style methods for weighted automata in verification-oriented settings.
In a nutshell, this chapter makes the following contributions:

1. We present a novel SMT-based active WFA learning algorithm (Section 6.2), which, if it terminates, is guaranteed to provide *minimal* WFAs. This guarantee of minimality on termination, which is semiring-agnostic, is a crucial advantage of our approach over $L^\star$-based methods where minimality guarantees only exist for very specific semirings (as we detail in Section 6.6).

2. We prove partial correctness of our algorithm and provide sufficient conditions for termination, which in particular apply to all finite semirings. As a corollary, we obtain provable guarantees for learning minimal NFAs.

3. At the heart of our algorithm is the first constraint-based WFA generation algorithm enabling SMT-based automata synthesis (Section 6.3), and we identify semirings for which the constraint systems are decidable (Section 6.4).

4. We provide an extensive experimental evaluation in Section 6.5, demonstrating that our algorithm is capable of learning numerous WFAs over both finite and infinite semirings. By comparing to a naive baseline as well as an existing WFA learning algorithm (which works for a restricted class of semirings), we show how our algorithm advances the state of the art in active learning for weighted automata over a variety of semirings.

We discuss related work in Section 6.6 and conclude in Section 6.7.

6.1 Semirings and Weighted Automata

In this section, we briefly recap preliminary material on weighted automata and weighted languages, which are parametric on a semiring.

Definition 6.1.1 (Semiring)**.** *A* semiring *is a tuple* $\langle \mathbb{S}, \oplus, \odot, 0_{\mathbb{S}}, 1_{\mathbb{S}} \rangle$*, where* $\mathbb{S}$ *is a* carrier *set,* $\langle \mathbb{S}, \oplus, 0_{\mathbb{S}} \rangle$ *is a commutative monoid,* $\langle \mathbb{S}, \odot, 1_{\mathbb{S}} \rangle$ *is a monoid, multiplication distributes over addition, i.e., for all* $s_1, s_2, s_3 \in \mathbb{S}$*:*

$$s_1 \odot (s_2 \oplus s_3) \;=\; (s_1 \odot s_2) \oplus (s_1 \odot s_3) \qquad (s_2 \oplus s_3) \odot s_1 \;=\; (s_2 \odot s_1) \oplus (s_3 \odot s_1)$$

Additionally, $0_{\mathbb{S}}$ *is annihilating, i.e., for all* $s \in \mathbb{S}$*, we have* $s \odot 0_{\mathbb{S}} = 0_{\mathbb{S}} = 0_{\mathbb{S}} \odot s$*.* $\triangle$

In this chapter, we primarily focus on the following semirings [DKV09, Sec. 2.1]:

1. The *Boolean semiring* $\mathbb{B} \triangleq \langle \{0,1\}, \vee, \wedge, 0, 1 \rangle$.

2. The *Tropical semiring* $\mathbb{T} \triangleq \langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0 \rangle$.

3. The family of *bounded Tropical semirings*, defined for a bound $b \in \mathbb{N}$ as $\mathbb{T}_b \triangleq \langle \{0,\ldots,b\} \cup \{\infty\}, \min, +_b, \infty, 0\rangle$,, where $s +_b s' \triangleq \infty$ whenever $s + s' > b$.

4. The *Bottleneck semiring* $\mathbb{Bot} \triangleq \langle \mathbb{N} \cup \{-\infty, \infty\}, \max, \min, -\infty, \infty\rangle$.

Below, to define weighted automata, we will use $\mathbb{S}$-valued matrices which we denote by $\mathcal{W}_{\mathbb{S}}(R \times C)$ where R and C are the sets indexing the rows and columns. When R (resp. C) is a singleton set (such as $\mathbb{1}$) the matrix is simply an $\mathbb{S}$-valued row (resp. column) vector. Formally, for a set X:

$$\mathcal{W}_{\mathbb{S}}(X) \triangleq \{v \mid v \colon X \to \mathbb{S}\}$$

We will use $\times$ for standard matrix multiplication of these objects. For the rest of the chapter, we fix a semiring $\langle \mathbb{S}, \oplus, \odot, 0_{\mathbb{S}}, 1_{\mathbb{S}}\rangle$, and a finite alphabet Σ.

Definition 6.1.2 (WFA)**.** *A* weighted finite automaton [DKV09] over $\mathbb{S}$ ($\mathbb{S}$-WFA) *is a tuple* $\mathcal{A} \triangleq \langle Q, \iota, \delta, \lambda\rangle$ *where* Q *is a finite nonempty set of states,* $\iota \in \mathcal{W}_{\mathbb{S}}(\mathbb{1} \times Q)$ *is the initial weight vector,* δ *is a* Σ*-indexed family of transition matrices* $\delta^a \in \mathcal{W}_{\mathbb{S}}(Q \times Q)$*, and* $\lambda \in \mathcal{W}_{\mathbb{S}}(Q \times \mathbb{1})$ *is the final weight vector.*
The semantics of a WFA are given by its language $[\![\mathcal{A}]\!](\iota) \colon \Sigma^\star \to \mathbb{S}$ *defined as:*

$$[\![\mathcal{A}]\!](v)(\varepsilon) \triangleq v \times \lambda \qquad [\![\mathcal{A}]\!](v)(a \cdot w) \triangleq [\![\mathcal{A}]\!](v \times \delta^a)(w)$$

Where clear we omit the initial weight vector from $[\![\mathcal{A}]\!](\iota)$ *writing simply* $[\![\mathcal{A}]\!]$. △

When $\mathbb{S}$ is clear from context, we simply write WFA instead of $\mathbb{S}$-WFA. WFAs recognize *weighted languages*, functions $\mathcal{L} \colon \Sigma^\star \to \mathbb{S}$. We call $[\![\mathcal{A}]\!](w)$ the *weight* $\mathcal{A}$ *assigns to* w. Note how for the Boolean semiring the definition of WFA coincides with that of nondeterministic finite automata (NFAs), and *weighted languages* over $\mathbb{B}$ are languages in the usual sense (cf. Section 2.1).

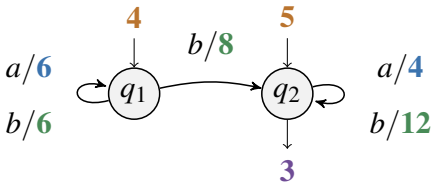
The set of all weighted languages is denoted by $\mathbb{S}\langle\langle\Sigma^\star\rangle\rangle$. A language $\mathcal{L}$ is *recognizable*, denoted $\mathcal{L} \in \mathbb{S}\langle\langle\Sigma^\star\rangle\rangle^{rec}$, if there is a WFA $\mathcal{A}$ recognizing $\mathcal{L}$, i.e., $[\![\mathcal{A}]\!] = \mathcal{L}$. A WFA $\mathcal{A}$ is *minimal*, if there does not exist a WFA $\mathcal{A}'$ with fewer states and $[\![\mathcal{A}]\!] = [\![\mathcal{A}']\!]$. We denote by $|\mathcal{L}|$ the *dimension of* $\mathcal{L} \in \mathbb{S}\langle\langle\Sigma^\star\rangle\rangle^{rec}$, which is the

number of states of a minimal automaton recognizing $\mathcal{L}$. Note that minimal WFAs are not unique in the same sense as deterministic finite automata — there might be different WFAs over the same number of states recognizing the same language. However, the number of states of a minimal automaton is uniquely determined.

Example 6.1.3. *Consider the following 2-state WFA* $\mathcal{A} \triangleq \langle \{q_1, q_2\}, \iota, \delta, \lambda \rangle$ *over the tropical semiring* $\mathbb{T}$ *and the alphabet* $\{a, b\}$*:*

$$\iota \triangleq \begin{pmatrix} 4 & 5 \end{pmatrix} \qquad \delta^a \triangleq \begin{pmatrix} 6 & \infty \\ \infty & 4 \end{pmatrix}$$

$$\delta^b \triangleq \begin{pmatrix} 6 & 8 \\ \infty & 12 \end{pmatrix} \qquad \lambda \triangleq \begin{pmatrix} \infty \\ 3 \end{pmatrix}$$



On the right, we depict the automaton as a state diagram and use colored weights to highlight the correspondence. $0_{\mathbb{S}}$*-valued transitions are omitted (and for the Tropical semiring* $\mathbb{T}$ *we have* $0_{\mathbb{S}} = \infty$*). The weight* $[\![\mathcal{A}]\!](ab)$ *that* $\mathcal{A}$ *assigns to ab is:*

$$[\![\mathcal{A}]\!](ab) = \iota \times \delta^a \times \delta^b \times \lambda = \begin{pmatrix} 4 & 5 \end{pmatrix} \begin{pmatrix} 6 & \infty \\ \infty & 4 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ \infty & 12 \end{pmatrix} \begin{pmatrix} \infty \\ 3 \end{pmatrix} = 21$$

6.2 Learning Algorithm

In this section, we give a high-level overview of our learning algorithm (shown in Algorithm 6.1), which will take as input a *teacher* holding a weighted language $\mathcal{L}$, and return a minimal WFA recognizing $\mathcal{L}$.

6.2.1 The Teacher

Classically, a *minimally adequate teacher* (Definition 2.3.1) in automata learning holds a (hidden) regular language $\mathcal{L}$, and answers two types of queries: membership queries (*mq*) returning whether a given string is part of the language, and equivalence queries (*eq*) determining whether a hypothesis model is correct, and returning a counterexample word if not. For a weighted (recognizable) language $\mathcal{L} \in \mathbb{S}\langle\langle \Sigma^\star \rangle\rangle^{rec}$, this concept is generalized into the quantitative setting: membership queries provide instead the *weight* that $\mathcal{L}$ assigns to a word. For consistency with previous work on

weighted automata learning, we too refer to these as membership queries.

Definition 6.2.1 (Teacher)**.** *A* teacher *(for $\mathcal{L}$) is a pair $\mathcal{T} \triangleq \langle mq, eq \rangle$, where:*

1. **Membership Queries** *of the type $mq\colon \Sigma^\star \to \mathbb{S}$ give the* weight *of a word in $\mathcal{L}$.*

2. **Equivalence Queries** *of the type $eq\colon \mathsf{WFA} \to \mathbb{1} + \Sigma^\star$ indicate that a hypothesis WFA is correct, or provide a counterexample witnessing a discrepancy.* △

We call (possibly infinite) nonempty sets $O \subseteq \Sigma^\star$ of words *observation sets*, which give rise to the straightforward notion of O-correct WFAs — WFAs that correctly recognize $\mathcal{L}$ when restricting to words in O:

Definition 6.2.2 (O-Correctness)**.** *A WFA $\mathcal{A}$ is O-correct w.r.t. $\mathcal{L}$, if:*

$$\forall w \in O. \quad [\![\mathcal{A}]\!](w) = \mathcal{L}(w) \qquad \triangle$$

6.2.2 The Main Loop of the Algorithm

Fix a teacher $\mathcal{T} \triangleq \langle mq, eq \rangle$ for $\mathcal{L}$ throughout this section. Our learning algorithm $\mathsf{learn}(\mathcal{T})$, depicted in Algorithm 6.1, takes $\mathcal{T}$ as input and, if it terminates, is guaranteed to produce a *minimal* WFA recognizing $\mathcal{L}$.

The main loop of the algorithm constructs new hypothesis automata using a subroutine getAutomaton that satisfies the following high-level specification.

Definition 6.2.3 (Specification of getAutomaton)**.** *Given access to membership queries via mq, the procedure* $\mathsf{getAutomaton}(mq, O, n)$*, when it terminates, maps each finite observation set O and $n \geq 1$ to either $\bullet \in \mathbb{1}$ or a hypothesis $\mathcal{H} \in \mathsf{WFA}$. The implementation of the procedure is* sound *if:*

1. $\mathsf{getAutomaton}(mq, O, n) = \bullet \in \mathbb{1}$ *iff there is no n-state O-correct WFA,*

2. $\mathsf{getAutomaton}(mq, O, n) = \mathcal{H} \in \mathsf{WFA}$ *implies $\mathcal{H}$ is an n-state O-correct WFA.*

Moreover, if it always terminates then we say it is complete. △

Algorithm 6.1: learn($\mathcal{T}$)

Data: A teacher $\mathcal{T} = \langle mq, eq \rangle$ for $\mathcal{L}$.
Result: Minimal WFA $\mathcal{H}$ recognizing $\mathcal{L}$.

```
1  O ← {ε}; n ← 1; H ← ⊥;
2  while ⊤ do
       // Loop Invariant: O is a finite observation set
       // and H ∈ WFA implies H is an n-state O-correct WFA
3      H ← getAutomaton(mq, O, n);
4      if H ∉ WFA then
           // No n-state O-correct WFA exists
5          n ← n + 1; continue
       // H is an n-state O-correct WFA
6      cex ← eq(H);
7      if cex ∉ Σ* then
           // H recognizes L
8          return H;
       // cex ∈ Σ* is a counterexample to H recognizing L
9      O ← O ∪ {cex};
```

Note that, in this section, getAutomaton is treated as a closed-box. Later, we will evaluate two distinct ways for implementing getAutomaton (cf. Section 6.3). Let us now explain how Algorithm 6.1 works. We first initialize the finite observation set $O \leftarrow \{\varepsilon\}$ and $n \leftarrow 1$ in l.1. The subsequent loop, on every iteration, invokes getAutomaton to determine whether an n-state O-correct WFA $\mathcal{H}$ exists (l. 3). If not, we increase n (l. 5). Otherwise, we ask the teacher whether $\mathcal{H}$ is a sought-after WFA recognizing $\mathcal{L}$ (l. 8). If so, we return $\mathcal{H}$ (l. 8). Otherwise, *cex* will be a counterexample — a word in $\Sigma^\star$ such that $[\![\mathcal{H}]\!](w) \neq \mathcal{L}(w)$ — which we add to the observation set O. This ensures that, indeed, we construct "increasingly correct" WFAs (w.r.t. the growing observation set O).

Based on the specification of getAutomaton, we consider partial correctness and sufficient termination criteria for the algorithm.

Theorem 6.2.4 (Partial Correctness of Algorithm 6.1)**.** *Let* getAutomaton *be sound. If* learn($\mathcal{T}$) *terminates, then*

$$\text{learn}(\mathcal{T}) = \mathcal{H} \qquad \textit{implies} \qquad \mathcal{H} \textit{ is a minimal WFA recognizing } \mathcal{L}\,.$$

Proof. If $\mathsf{learn}(\mathcal{T}) = \mathcal{H}$, then Algorithm 6.1 terminates in l. 8. Hence, $\mathcal{H}$ recognizes $\mathcal{L}$ by Definition 6.2.1.2. It remains to show that $\mathcal{H}$ is minimal. We proceed by contradiction. For that, assume $\mathcal{H}$ is *not* minimal, i.e., the final value of n in Algorithm 6.1 exceeds $|\mathcal{L}|$. Consequently, some call to $\mathsf{getAutomaton}(mq, O, |\mathcal{L}|)$ returned $\bullet \in \mathbb{1}$. By Definition 6.2.3, there is thus no $|\mathcal{L}|$-state O-correct WFA, which is a contradiction since such a WFA exists for all $O \subseteq \Sigma^\star$. □

Next, we provide a sufficient termination condition for our learning algorithm.

Theorem 6.2.5 (Sufficient Termination Condition for Algorithm 6.1)**.** *Let* getAutomaton *be sound and complete. If, for all $n \geq 1$, there does not exist an infinite sequence $(O_1, \mathcal{H}_1), (O_2, \mathcal{H}_2), \ldots$ of n-state O_i-correct WFAs $\mathcal{H}_i$ such that*

$$O_1 \subsetneq O_2 \subsetneq \ldots,$$

then $\mathsf{learn}(\mathcal{T})$ *terminates.*

Proof. First observe that, for every n encountered during execution, Algorithm 6.1 eventually reaches l. 5 or l. 8 since, otherwise, the algorithm would produce an infinite chain of "increasingly O-correct" n-state WFAs.

For all $n < |\mathcal{L}|$, it reaches l. 5 since terminating in l. 8 would contradict Theorem 6.2.4: For $n < |\mathcal{L}|$, there is no n-state WFA recognizing $\mathcal{L}$. Consequently, Algorithm 6.1 eventually reaches $n = |\mathcal{L}|$. In that case, it eventually terminates in l. 8 because terminating in l. 5 contradicts Definition 6.2.3: For all O, there exists an $|\mathcal{L}|$-state O-correct WFA. □

Corollary 6.2.6 (Termination for Finite Semirings)**.** *Take* getAutomaton *to be sound and complete. If $\mathbb{S}$ is finite, then* $\mathsf{learn}(\mathcal{T})$ *terminates.*

Proof. The condition from Theorem 6.2.5 is trivially satisfied since, for every $n \geq 1$, there exist only finitely many n-state WFAs over a finite semiring. □

Remark 6.2.7 (Beyond Finite Semirings)**.** *Since a sound and complete* getAutomaton *can be implemented in a brute-force manner for all finite semirings, it follows that Algorithm 6.1 is correct and terminates for all finite semirings. The*

reader might wonder if one can prove soundness and completeness for more general semirings, since finiteness is a stringent restriction. One might have to impose further restrictions on the sequence $\mathcal{H}_1, \mathcal{H}_2, \ldots$ of hypotheses to guarantee termination beyond *finite semirings. Indeed, if $\mathbb{S}$ is finite, then for fixed n there are only finitely many n-state WFA up to isomorphism (finitely many choices of initial/final weights and transition weights). Hence no infinite sequence $\mathcal{H}_1$, $\mathcal{H}_2$, ... exists at all.*

If $\mathbb{S}$ is not finite, there are infinitely many distinct n-state WFA, so a pure cardinality argument disappears, and completeness is much more nuanced. This is in contrast with $\mathrm{L}^\star$ *algorithms where correctness based on Hankel matrices provides a relation between the state spaces of subsequent hypotheses $\mathcal{H}_1$, $\mathcal{H}_2$, ... appearing in the algorithm, and for which algebraic properties such as the semiring being a field or Noetherian can then be used to argue termination. As we will describe in Section 6.6, the study of termination for these* $\mathrm{L}^\star$*-like algorithms is subtle and has resulted in a variety of recent results.*

In Section 6.5, we will see that even though we do not guarantee termination, our implementation is capable of learning minimal WFAs over several infinite semirings, even for examples in which the existence of complete learning algorithms is open (e.g., the Tropical or the Bottleneck semiring).

6.3 Computing *n*-State *O*-Correct WFAs

In this section, we present two procedures — a naive baseline as well as a more sophisticated technique inspired by a coalgebraic perspective on WFAs — to implement getAutomaton as specified in Definition 6.2.3. Both procedures reduce the problem of determining the existence of *n*-state *O*-correct WFAs to a *satisfiability problem of equality constraints* over the given semiring $\mathbb{S}$.

We will show that sound and complete implementations of getAutomaton exist if the underlying semiring is *decidable* — these are semirings for which the satisfiability of *finite equation systems* is decidable. Put more formally, finite systems Φ of equations over $\mathbb{S}$ adhere to the grammar:

$$\Phi \longrightarrow t = t \mid \Phi \wedge \Phi \qquad t \longrightarrow s \in \mathbb{S} \mid x \mid t \oplus t \mid t \odot t$$

Where x is taken from some set of variables. This yields the following notion:

Definition 6.3.1 (Decidable Semiring)**.** *We say that a semiring* $\langle \mathbb{S}, \oplus, \odot, 0_{\mathbb{S}}, 1_{\mathbb{S}} \rangle$ *is* decidable*, if there is a computable function* $\mathsf{SAT}(\Phi)$ *mapping each finite system* Φ *of equations to either (i)* $\bullet \in \mathbb{1}$ *iff* Φ *has no solution, or (ii) a function* model *from the variables in* Φ *to values in* $\mathbb{S}$*, representing a satisfying assignment for* Φ*.* △

Every finite semiring is trivially decidable by brute-forcing all possible solutions to Φ. Interestingly, various *infinite* semiring are decidable as well by reducing to decidable arithmetical theories. All the example semirings we consider are decidable. We will provide two distinct ways of obtaining the following result:

Theorem 6.3.2. *For every decidable semiring, there is a sound and complete implementation of* getAutomaton*.*

Proof. Since $\mathbb{S}$ is decidable, finite systems of equations Φ over $\mathbb{S}$ can be solved effectively. We can therefore implement $\mathsf{getAutomaton}(mq, n, O)$ by encoding a general n-state WFA whose semantics agree with mq on O. The resulting implementation is detailed in section 6.3.1. □

For the semirings considered in this chapter, we employ SMT solving over decidable theories to find solutions to said systems of equations, or for proving their absence. The respective encodings are presented in Section 6.4.

6.3.1 A Naive Baseline Implementation

Let $n \geq 1$ and let O be a nonempty finite observation set. Our naive baseline implementation of $\mathsf{getAutomaton}(mq, n, O)$ is derived directly from the semantics of WFAs presented in Section 6.1: We treat the initial weight vector ι, the transition matrices δ^a, and the final weight vector λ — all of which having the fixed dimension

n — as unknowns and solve the finite equation system:

$$\Phi(n,O) \triangleq \bigwedge_{a_1\ldots a_k \in O} mq(a_1\ldots a_k) = \iota \times \delta^{a_1} \times \ldots \times \delta^{a_k} \times \lambda$$

It is immediate that $\mathsf{SAT}(\Phi(n,O)) = \bullet \in \mathbb{1}$ iff there is no n-state O-correct WFA, and that, otherwise, every model of Φ gives rise to an n-state O-correct WFA. Hence, if the semiring $\mathbb{S}$ is decidable, Theorem 6.3.2 follows.

Example 6.3.3. *For the $\mathbb{T}$-WFA of Example 6.1.3, and $O = \{ab\}$:*

$$\Phi(2,O) \triangleq 21 = \begin{pmatrix} \iota_1 & \iota_2 \end{pmatrix} \times \begin{pmatrix} \delta^a_{1,1} & \delta^a_{1,2} \\ \delta^a_{2,1} & \delta^a_{2,2} \end{pmatrix} \times \begin{pmatrix} \delta^b_{1,1} & \delta^b_{1,2} \\ \delta^b_{2,1} & \delta^b_{2,2} \end{pmatrix} \times \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}$$

We can see that the formula encodes directly the correctness of the automaton's weights, and therefore its language, w.r.t. the word ab.

6.3.2 An Optimized Implementation via State Languages

The central idea for our main approach is to exploit a coalgebraic perspective on the semantics of WFAs [BMS13] which enables a compositional view of the language of an automaton as a linear combination of languages of its individual states. Formally, for $\mathcal{A} = \langle Q, \iota, \delta, \lambda \rangle$, we define $[\![\mathcal{A}, q]\!]$, the *state language* of $q \in Q$ as:

$$[\![\mathcal{A},q]\!](\varepsilon) \triangleq \lambda(q) \qquad [\![\mathcal{A},q]\!](aw) \triangleq \delta^a_{q,q_1} \odot [\![\mathcal{A},q_1]\!](w) \oplus \ldots \oplus \delta^a_{q,q_n} \odot [\![\mathcal{A},q_n]\!](w)$$

State languages are naturally captured recursively, i.e., if $w = \varepsilon$, then $[\![\mathcal{A},q]\!](\varepsilon)$ is q's output weight $\lambda(q)$ since there are no symbols to be consumed. Otherwise, if $w = aw'$, then $[\![\mathcal{A},q]\!](aw')$ is obtained from "consuming" a, and proceeding by running $\mathcal{A}$ on w' from the successors of q. Given the state languages $[\![\mathcal{A},q]\!](w)$ at every state $q \in Q$, we obtain the weight $[\![\mathcal{A}]\!](w)$ by taking the initial weights given by ι into account. This yields the following relationship with the language $[\![\mathcal{A}]\!]$ of $\mathcal{A}$:

Theorem 6.3.4. *Let $\mathcal{A} = \langle Q, \iota, \delta, \lambda \rangle$ be a WFA with $Q = \{q_1, \ldots, q_n\}$. Then:*

$$\forall w \in \Sigma^\star. \quad [\![\mathcal{A}]\!](w) = \iota(q_1) \odot [\![\mathcal{A}, q_1]\!](w) \oplus \ldots \oplus \iota(q_n) \odot [\![\mathcal{A}, q_n]\!](w)$$

Proof. We show a more general fact that, for every initial vector v and word w:

$$[\![\mathcal{A}]\!](v)(w) = v(q_1) \odot [\![\mathcal{A}, q_1]\!](w) \oplus \ldots \oplus v(q_n) \odot [\![\mathcal{A}, q_n]\!](w)$$

This follows by induction on the length of w. For $w = \varepsilon$, the claim is precisely the matrix product $v \times \lambda$, since $[\![\mathcal{A}, q_i]\!](\varepsilon) = \lambda(q_i)$. For $w = aw'$, we calculate:

$$\begin{aligned}
[\![\mathcal{A}]\!](v)(aw') &= [\![\mathcal{A}]\!](v \times \delta^a)(w') && \text{(WFA semantics)} \\
&= (v \times \delta^a)(q_1) \odot [\![\mathcal{A}, q_1]\!](w') \oplus \ldots \oplus (v \times \delta^a)(q_n) \odot [\![\mathcal{A}, q_n]\!](w') \\
& && \text{(induction hypothesis)} \\
&= \Big(\big(v(q_1) \odot \delta^a_{1,1} \oplus \ldots \oplus v(q_n) \odot \delta^a_{n,1} \big) \odot [\![\mathcal{A}, q_1]\!](w') \Big) \oplus \ldots \oplus \\
&\quad \Big(\big(v(q_1) \odot \delta^a_{1,n} \oplus \ldots \oplus v(q_n) \odot \delta^a_{n,n} \big) \odot [\![\mathcal{A}, q_n]\!](w') \Big) \\
& && \text{(expanding the matrix products)} \\
&= v(q_1) \odot \Big(\delta^a_{1,1} \odot [\![\mathcal{A}, q_1]\!](w') \oplus \ldots \oplus \delta^a_{1,n} \odot [\![\mathcal{A}, q_n]\!](w') \Big) \oplus \ldots \oplus \\
&\quad v(q_n) \odot \Big(\delta^a_{n,1} \odot [\![\mathcal{A}, q_1]\!](w') \oplus \ldots \oplus \delta^a_{n,n} \odot [\![\mathcal{A}, q_n]\!](w') \Big) \\
& && \text{(semiring laws)} \\
&= v(q_1) \odot [\![\mathcal{A}, q_1]\!](aw') \oplus \ldots \oplus v(q_n) \odot [\![\mathcal{A}, q_n]\!](aw') \\
& && \text{(state-language semantics)}
\end{aligned}$$

Taking $v = \iota$ proves the result. □

Example 6.3.5. *For automaton $\mathcal{A}$ from Example 6.1.3 we compute below $[\![\mathcal{A}]\!](ab)$ through its state languages.*

$$[\![\mathcal{A}, q_1]\!](\varepsilon) \triangleq \lambda(q_1) = \infty \qquad [\![\mathcal{A}, q_2]\!](\varepsilon) \triangleq \lambda(q_2) = 3$$

$$[\![\mathcal{A},q_1]\!](b) = \delta^b_{1,1} \odot [\![\mathcal{A},q_1]\!](\varepsilon) \oplus \delta^b_{1,2} \odot [\![\mathcal{A},q_2]\!](\varepsilon) \;=\; 6 \odot \infty \oplus\; 8 \odot 3 = 11$$

$$[\![\mathcal{A},q_2]\!](b) = \delta^b_{2,1} \odot [\![\mathcal{A},q_1]\!](\varepsilon) \oplus \delta^b_{2,2} \odot [\![\mathcal{A},q_2]\!](\varepsilon) \;= \infty \odot \infty \oplus 12 \odot 3 = 15$$

$$[\![\mathcal{A},q_1]\!](ab) = \delta^a_{1,1} \odot [\![\mathcal{A},q_1]\!](b) \oplus \delta^a_{1,2} \odot [\![\mathcal{A},q_2]\!](b) = 6 \odot 11 \oplus \infty \odot 15 = 17$$

$$[\![\mathcal{A},q_2]\!](ab) = \delta^a_{2,1} \odot [\![\mathcal{A},q_1]\!](b) \oplus \delta^a_{2,2} \odot [\![\mathcal{A},q_2]\!](b) = \infty \odot 11 \oplus 4 \odot 15 = 19$$

$$[\![\mathcal{A}]\!](ab) = \iota(q_1) \odot [\![\mathcal{A},q_1]\!](ab) \oplus \iota(q_2) \odot [\![\mathcal{A},q_2]\!](ab) = 4 \odot 17 \oplus 5 \odot 19 = 21$$

In our approach to learning we will explore Theorem 6.3.4 to derive an encoding of a more state-local characterization of the existence of n-state O-correct WFAs. Our experiments in Section 6.5 reveal this characterization to be highly beneficial when compared to the naive baseline from the previous section. Towards formalizing this characterization, we introduce:

Definition 6.3.6 (O-Witness)**.** *Let O be an observation set. An O-*witness *(of $\mathcal{L}$) is a triple $\mathcal{W} \triangleq \langle \mathcal{F}, \iota, \delta \rangle$, where:*

1. *$\mathcal{F} = (f_i)_{i=1}^n$ is an indexed family of O-*generators *with type* $\mathsf{suffixes}(O) \to \mathbb{S}$*,*
2. *δ is a Σ-indexed family of matrices $\delta^a \colon \mathcal{W}_{\mathbb{S}}(\mathcal{F} \times \mathcal{F})$ satisfying:*

$$\underbrace{\forall f_i \in \mathcal{F}.\ \forall aw \in \mathsf{suffixes}(O).\quad f_i(aw) = \delta^a_{i,1} \odot f_1(w) \oplus \ldots \oplus \delta^a_{i,n} \odot f_n(w)}_{\delta \text{ stabilizes } \mathcal{F} \text{ for } O}$$

3. *$\iota \colon \mathcal{W}_{\mathbb{S}}(1 \times \mathcal{F})$ is a weight vector satisfying:*

$$\underbrace{\forall w \in O.\quad \mathcal{L}(w) = \iota_1 \odot f_1(w) \oplus \ldots \oplus \iota_n \odot f_n(w)}_{\iota \text{ spans } \mathcal{L} \text{ in } \mathcal{F} \text{ for } O}$$

△

Notice that we do not require O to be finite. We call the number n of generators in $\mathcal{F}$ the *dimension of $\mathcal{W}$* and denote it by $|\mathcal{W}|$. Intuitively, they correspond to the state languages in a WFA, with the imposed constraints on $\mathcal{F}$, δ, and ι making sure

that, when restricting to words in O, these state languages are indeed captured by a WFA-structure. Hence, by Theorem 6.3.4, there is a one-to-one correspondence between witnesses and WFAs: Every n-dimensional O-witness yields an n-state O-correct automaton, and vice versa. It is in this sense that $\mathcal{W}$ *witnesses* the existence of the sought-after WFAs. Put formally:

Theorem 6.3.7 (WFA-Witness Correspondence)**.** *Let $n \geq 1$ and let O be an observation set. We have:*

1. *If $\mathcal{W} = \langle \mathcal{F}, \iota, \delta \rangle$ is an n-dimensional O-witness, then $\mathcal{A}_{\mathcal{W}} = \langle \mathcal{F}, \iota, \delta, \lambda \rangle$, where $\lambda(f_i) = f_i(\varepsilon)$ is an n-state O-correct WFA.*

2. *If $\mathcal{A} = \langle \{q_1, \ldots, q_n\}, \iota, \delta, \lambda \rangle$ is an O-correct WFA, then the triple $\mathcal{W}_{\mathcal{A}} = \langle (f_i)_{i=1}^n, \iota, \delta \rangle$ is an n-dimensional O-witness, where $f_i \colon \mathsf{suffixes}(O) \to \mathbb{S}$ are uniquely determined by $f_i(\varepsilon) = \lambda(q_i)$ and $f_i(aw) \triangleq \delta_{i,1}^a \odot f_1(w) \oplus \cdots \oplus \delta_{i,n}^a \odot f_n(w)$, for all $aw \in \mathsf{suffixes}(O)$.*

Proof. For 1, we first show that each generator is the corresponding state language of $\mathcal{A}_{\mathcal{W}}$ on $\mathsf{suffixes}(O)$:

$$\forall w \in \mathsf{suffixes}(O)\,. \quad [\![\mathcal{A}_{\mathcal{W}}, f_i]\!](w) = f_i(w)$$

This follows by induction on the length of w, simultaneously for all i. At ε, it holds by the definition $\lambda(f_i) = f_i(\varepsilon)$. For aw, it follows by unfolding the state language, applying the induction hypothesis to w, and using the stabilization condition of the witness. Notice that w remains in $\mathsf{suffixes}(O)$ because this set is suffix-closed. Hence, for every $w \in O$, Theorem 6.3.4 and the spanning condition ensure $\mathcal{A}_{\mathcal{W}}$ is O-correct:

$$[\![\mathcal{A}_{\mathcal{W}}]\!](w) = \iota_1 \odot f_1(w) \oplus \ldots \oplus \iota_n \odot f_n(w) = \mathcal{L}(w)$$

For 2, suffix-closure also makes the displayed recursion well-defined: induction on word length uniquely determines every f_i on $\mathsf{suffixes}(O)$. Its base case and recursive step are exactly those of $[\![\mathcal{A}, q_i]\!]$, so $f_i(w) = [\![\mathcal{A}, q_i]\!](w)$ throughout this

domain. The recursive step immediately gives the stabilization condition. Finally, for every $w \in O$, O-correctness and Theorem 6.3.4 yield the spanning condition:

$$\mathcal{L}(w) = [\![\mathcal{A}]\!](w) = \iota_1 \odot f_1(w) \oplus \ldots \oplus \iota_n \odot f_n(w)$$

Therefore $\mathcal{W}_\mathcal{A}$ is an n-dimensional O-witness. □

Notice that observation sets being nonempty implies $\varepsilon \in \mathsf{suffixes}(O)$, which is why the output weight vector λ can be readily read off the weights $f_i(\varepsilon)$.

It is immediate then that the existence of n-state O-correct automata can be characterized via the existence of n-dimensional O-witnesses:

Corollary 6.3.8 (Existence of O-Correct WFA)**.** *Let $n \geq 1$ and let O be an observation set. The following statements are equivalent:*

1. *There exists an n-dimensional O-witness $\mathcal{W} = (\mathcal{F}, \iota, \delta)$.*
2. *There exists an n-state O-correct WFA $\mathcal{A}$.*

Proof. By the correspondence of WFAs and witnesses of Theorem 6.3.7. □

We now arrive at our system of equations $\Phi(n, O)$ depicted in Figure 6.1. For all $n \geq 1$ and finite observation sets O, $\Phi(n, O)$ has a solution iff there exists an n-dimensional O-witness. Moreover, every model of $\Phi(n, O)$ gives rise to such a witness $\mathcal{W}$ and therefore also to an n-state O-correct automaton $\mathcal{A}_\mathcal{W}$ by Theorem 6.3.7.

$$\Phi(n, O) \quad \triangleq \quad \underbrace{\bigwedge_{i=1}^{n} \bigwedge_{aw \in \mathsf{suffixes}(O)} f_i^{aw} = (\delta_{i,1}^a \odot f_1^w) \oplus \ldots \oplus (\delta_{i,n}^a \odot f_n^w)}_{\delta \text{ stabilizes } \mathcal{F} \text{ for } O \text{ (Definition 6.3.6.2)}}$$
$$\wedge \underbrace{\bigwedge_{w \in O} mq(w) = (\iota_1 \odot f_1^w) \oplus \ldots \oplus (\iota_n \odot f_n^w)}_{\iota \text{ spans } \mathcal{L} \text{ for } O \text{ (Definition 6.3.6.3)}}$$

Figure 6.1: Equation system $\Phi(O, n)$ characterizing the existence of n-state O-correct WFAs. Here $(f_i^w)_{i \in n, w \in \mathsf{suffixes}(O)}$, $(\delta_{i,j}^a)_{a \in \Sigma, (i,j) \in n \times n}$, and $(\iota_i)_{i \in n}$ are $\mathbb{S}$-valued variables.

The crucial observation is that, since O is finite, so is $\mathsf{suffixes}(O)$ and thus all potential generators f_i range over a finite domain. The existence of suitable generators f_i, transition matrices δ^a, and initial weights ι can thus be expressed in terms of a system of equations, where we introduce a variable f_i^w for each application of some $w \in \mathsf{suffixes}(O)$ to some generator f_i.

Example 6.3.9. *For the $\mathbb{T}$-WFA of Example 6.1.3, and $O = \{ab\}$ we have that:*

$$\begin{aligned}\Phi(2,O) \triangleq\ & 21 = (\iota_1 \odot f_1^{ab}) \oplus (\iota_2 \odot f_2^{ab}) \\ & \wedge\ f_1^{ab} = (\delta^a_{1,1} \odot f_1^b) \oplus (\delta^a_{1,2} \odot f_2^b)\ \wedge\ f_1^b = (\delta^b_{1,1} \odot f_1^\varepsilon) \oplus (\delta^b_{1,2} \odot f_2^\varepsilon) \\ & \wedge\ f_2^{ab} = (\delta^a_{2,1} \odot f_1^b) \oplus (\delta^a_{2,2} \odot f_2^b)\ \wedge\ f_2^b = (\delta^b_{2,1} \odot f_1^\varepsilon) \oplus (\delta^b_{2,2} \odot f_2^\varepsilon)\end{aligned}$$

As Examples 6.3.3 and 6.3.9 illustrate, the optimized encoding introduces more equations and more variables than the naive baseline. Counterintuitively, this apparent overhead is precisely what enables its scalability. As the observation set O grows, the intermediate variables expose structure shared between observations with matching suffixes, which the SMT solver can exploit.

Example 6.3.10. *Suppose O grows from $\{ab\}$ to $\{ab, aab\}$. The naive encoding adds a single, but more complex, equation:*

$$mq(aab) \;=\; \begin{pmatrix}\iota_1 & \iota_2\end{pmatrix} \times \begin{pmatrix}\delta^a_{1,1} & \delta^a_{1,2} \\ \delta^a_{2,1} & \delta^a_{2,2}\end{pmatrix} \times \begin{pmatrix}\delta^a_{1,1} & \delta^a_{1,2} \\ \delta^a_{2,1} & \delta^a_{2,2}\end{pmatrix} \times \begin{pmatrix}\delta^b_{1,1} & \delta^b_{1,2} \\ \delta^b_{2,1} & \delta^b_{2,2}\end{pmatrix} \times \begin{pmatrix}\lambda_1 \\ \lambda_2\end{pmatrix}$$

The optimized encoding, on the other hand, adds one spanning equation together with two stabilization equations for the new variables f_1^{aab}, f_2^{aab}:

$$\begin{aligned}mq(aab) \;&=\; (\iota_1 \odot f_1^{aab}) \oplus (\iota_2 \odot f_2^{aab}) \\ f_1^{aab} = (\delta^a_{1,1} \odot f_1^{ab}) \oplus (\delta^a_{1,2} \odot f_2^{ab})\ &\wedge\ f_2^{aab} = (\delta^a_{2,1} \odot f_1^{ab}) \oplus (\delta^a_{2,2} \odot f_2^{ab})\end{aligned}$$

Crucially, the variables f_1^{ab}, f_2^{ab} introduced when ab was first added to O are reused, so the shared suffix ab is reflected directly in shared subterms of the equation system.

6.4 Deciding Equation Systems via SMT Solving

Next, we discuss how to solve the equation systems, as defined in the beginning of Section 6.3, for the semirings considered in this chapter. We do this by reducing the problem of computing $\mathsf{SAT}(\Phi)$ to satisfiability problems over decidable theories, which enables using SMT solvers. Every system Φ of equations over the Boolean semiring directly corresponds to a propositional satisfiability problem. The encoding is thus straightforward. Let us consider the remaining semirings:

6.4.1 The Tropical Semiring

Consider solving equation systems Φ over the tropical semiring:

$$\mathbb{T} \triangleq \langle \mathbb{S}, \oplus, \odot, 0_{\mathbb{S}}, 1_{\mathbb{S}} \rangle \triangleq \langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0 \rangle$$

We reduce to a satisfiability problem over the decidable theory QF-LIA of quantifier-free linear integer arithmetic. Variables are encoded from $\mathbb{N} \cup \{\infty\}$ into $\{-1, 0, 1, 2, \dots\} \subseteq \mathbb{Z}$, where -1 is used as a special value representing ∞. The semiring operations are then encoded as:

$$t \oplus t' \triangleq \textbf{if } \underbrace{t = -1 \vee t' = -1}_{t \text{ or } t' \text{ equals } \infty} \textbf{ then } \underbrace{max(t,t')}_{t \text{ if } t' = \infty \text{ else } t'} \textbf{ else } \underbrace{min(t,t')}_{\text{usual minimum in } \mathbb{N}}$$

$$t \odot t' \triangleq \textbf{if } \underbrace{t = -1 \vee t' = -1}_{t \text{ or } t' \text{ equals } \infty} \textbf{ then } \underbrace{-1}_{t+t'=\infty} \textbf{ else } \underbrace{(t + t')}_{\text{usual sum in } \mathbb{N}}$$

where both *max* and *min* are encoded via if-then-else in the obvious way.

6.4.2 The Bounded Tropical Semirings

Let $b \in \mathbb{N}$ and consider the b-bounded tropical semiring:

$$\mathbb{T}_b \triangleq \langle \mathbb{S}, \oplus, \odot, 0_{\mathbb{S}}, 1_{\mathbb{S}} \rangle \triangleq \langle \{0, \dots, b\} \cup \{\infty\}, \min, +_b, \infty, 0 \rangle$$

We consider two encodings for solving systems Φ of equations over that semiring: A QF-LIA encoding similar to the previous section and an encoding QF-BV into the quantifier-free theory of fixed-size bit vectors. Both encode the domain $\{0, \dots, b\} \cup$

$\{\infty\}$ into the finite set $\{0,\ldots,b,b+1\}$, where $b+1$ represents ∞.

The QF-LIA **Encoding** Since $b+1$ represents ∞, we straightforwardly let:

$$t \oplus t' \triangleq min(t,t') \qquad t \odot t' \triangleq min((t+t'),b+1)$$

The QF-BV **Encoding** We map the integer domain $\{0,\ldots,b,b+1\}$ into its (unsigned) binary number representation in *fixed-size bit vectors* of appropriate size l, and use fixed-sized bit vector addition and if-then-else to analogously define:

$$t \oplus t' \triangleq min(t,t') \qquad t \odot t' \triangleq min((t+t'),b+1)$$

We must, however, take care: While just $\lceil log_2(b+2)\rceil$ bits are sufficient to represent $\{0,\ldots,b,b+1\}$, *intermediate* results from the non-truncated addition $t+t'$ might overflow. The biggest value we must account for is thus $(b+1)+(b+1)=2(b+1)$, therefore needing to represent $2(b+1)+1=2b+3$ values, and so $l=\lceil log_2(2b+3)\rceil$ bits are sufficient to represent all values while avoiding overflows.

6.4.3 The Bottleneck Semiring

Consider the *bottleneck* semiring:

$$\mathbb{Bot} \triangleq \langle \mathbb{S},\oplus,\odot,0_{\mathbb{S}},1_{\mathbb{S}}\rangle \triangleq \langle \mathbb{N}\cup\{-\infty,\infty\},\max,\min,-\infty,\infty\rangle$$

We encode the carrier set of $\mathbb{N}\cup\{-\infty,\infty\}$ into $\{-2,-1,0,1,2,\ldots\}\subseteq\mathbb{Z}$, using -2 to represent $-\infty$ and -1 for ∞. The operations are then encoded in a similar fashion as the tropical operations, accounting for the special infinity values, and minimizing/maximizing otherwise:

$$\begin{aligned} t \oplus t' \triangleq\ & \textbf{if } \underbrace{t=-1 \vee t'=-1}_{t \text{ or } t' \text{ equals } \infty} \textbf{ then } \underbrace{-1}_{\max(t,t')=\infty} \\ & \textbf{else if } \underbrace{t=-2}_{t \text{ equals } -\infty} \textbf{ then } \underbrace{t'}_{\max(-\infty,t')=t'} \\ & \textbf{else if } \underbrace{t'=-2}_{t' \text{ equals } -\infty} \textbf{ then } \underbrace{t}_{\max(t,-\infty)=t} \textbf{ else } \underbrace{\max(t,t')}_{\max \text{ in } \mathbb{N}} \end{aligned}$$

$$t \odot t' \triangleq \textbf{if } \underbrace{t = -2 \vee t' = -2}_{t \text{ or } t' \text{ equals } -\infty} \textbf{ then } \underbrace{-2}_{\min(t,t') = -\infty}$$
$$\textbf{else if } \underbrace{t = -1}_{t \text{ equals } \infty} \textbf{ then } \underbrace{t'}_{\min(\infty,t') = t'}$$
$$\textbf{else if } \underbrace{t' = -1}_{t' \text{ equals } \infty} \textbf{ then } \underbrace{t}_{\min(t,\infty) = t} \textbf{ else } \underbrace{\min(t,t')}_{\min \text{ in } \mathbb{N}}$$

Remark 6.4.1 (Other Semirings)**.** *The semirings presented above admit linear encodings (*QF-BV*,* QF-LIA*) of our equation systems. While this tends to provide performance benefits at the SMT level, semirings such as the Reals, Viterbi, or Probability semirings (see, e.g., [Bat+22, Table 1]) could equally be considered due to their decidable, nonlinear (*QF-NRA*) theory. By Theorem 6.3.2,* getAutomaton *remains complete and O-correct automata can be learned. We view this as a foundation for tackling the long-standing open problem of learning probabilistic automata, and leave to future work a systematic exploration leveraging, e.g., approximate SMT solving [GAC12] or incremental linearization techniques [Cim+18].*

6.5 Experimental Evaluation

In this section we evaluate our algorithm through a wide range of experiments. We start by describing our prototype implementation, and move on to pose research questions on its performance. We develop a comprehensive set of benchmarks to determine the impact of performance optimizations, as well as the algorithm's positioning relative to the naive baseline and a state-of-the-art WFA learning algorithm.

6.5.1 Implementation

To enable an extensive experimental evaluation, we have implemented both Algorithm 6.1 with the naive baseline from Section 6.3.1 (referred to as NAIVE) and with the state-language-based encoding from Section 6.3.2 (referred to as **S**MT-based **W**eighted **A**utomata **L**earning algorithm (SWAL)) in OCaml.

Our implementation supports all semirings discussed in this chapter (Boolean, Tropical, bounded Tropical, and Bottleneck), and uses Z3 [DB08] as a back-end solver. Equation systems over these semirings are decided via the encodings of

Section 6.4. In the following, we discuss our tool's use of incremental SMT solving and how we implement teachers.

Incremental SMT Solving When the observation set O maintained by Algorithm 6.1 grows (for fixed n), the respective equation systems consist of more and more conjuncts for both NAIVE and SWAL. This renders our approach suitable for *incremental* SMT solving. Whether incremental SMT solving is enabled or not can be controlled via a flag. We evaluate the effect of incremental solving in Section 6.5.3.

Teachers Our benchmarks aim to learn a given *target WFA* $\mathcal{A}$, which is known to the teacher but not to the learner. We implement its teacher $\mathcal{T}_\mathcal{A} \triangleq \langle mq, eq \rangle$ as follows: Membership queries $mq(w)$ are implemented straightforwardly by computing $\mathcal{A}$'s semantics $[\![\mathcal{A}]\!](w)$. Since equivalence queries are generally undecidable [ABK22], we follow the standard procedure in automata learning [ASV10; ARP13; FJV16; Fer+21] and resort to *partial equivalence checks*: We let $eq(\mathcal{H}) \triangleq \bullet \in \mathbb{1}$ iff $[\![\mathcal{A}]\!](w) = [\![\mathcal{H}]\!](w)$ for the first N words in $\Sigma^\star$ w.r.t. a fixed lexicographic order on the symbols, and we let $eq(\mathcal{H}) \triangleq w$ only if w is the first counterexample found under these N words. For our experiments, we let $N \triangleq 5000 \cdot |\mathcal{A}| \cdot |\Sigma|$. We have found that this provides a vast cover of reachable automaton behavior[1], while remaining computationally tractable. Developing more advanced equivalence testing algorithms for weighted automata remains an interesting open area of research, with existing techniques such as the Wp-Method [Fuj+91] being largely motivated for the Boolean setting.

6.5.2 Research Questions and Methodology

We have devised three research questions (RQs) for evaluating our algorithms.

RQ1 What is the impact of incremental SMT solving and bit vector encodings?

RQ2 How does SWAL perform compared to the NAIVE algorithm?

RQ3 How does SWAL perform compared to the WL* algorithm?

RQ1 forms an *ablation study* aiming at determining the impact of incremental SMT solving and alternative encodings. RQ2 enables evaluating whether SWAL is

[1] For the automata learned, on average, counterexamples were found only in the first 5.02% of this test budget, with over 96.7% of runs utilizing $< 50\%$ of it.

indeed superior to the naive baseline NAIVE. In RQ3, WL* refers to a state-of-the-art WFA learning algorithm [Van+20] for finite semirings, or principal ideal domains. For the semirings considered in this chapter, WL* thus produces (not necessarily minimal) WFAs over the Boolean and the bounded Tropical semiring.

Benchmarks Answering our research questions requires a diverse set of benchmarks, covering a range of different WFAs for all considered semirings. Classically, one would rely on existing realistic datasets [Nei+19] for automata learning evaluations. The line of work in weighted automata learning has not yet contributed such a dataset. Therefore, we devised over 5000 randomly generated benchmarks.

Each run was allocated a memory limit of 8GB, and a time limit of 2 hours. Below we describe the range of parameters considered for these experiments. For all semirings considered in this chapter, we generate target automata with state sizes ranging from 1 to 12, and alphabet sizes ranging from 2 to 50 symbols.

For $\mathbb{B}$-WFAs, we use a well-studied family of minimal NFAs [DLT01, Proposition 6.2], which is parametric in the number of states and the alphabet size. This yields our scalable set of target WFAs over the Boolean semiring.

For $\mathbb{T}$-WFAs (both unbounded and bounded) and $\mathbb{Bot}$-WFAs, we generated the target WFAs randomly. Generating random *minimal* automata is, however, famously hard [TV05], with heuristics only known for DFAs and NFAs over the Boolean semiring. Exact algorithms would rely on generating random automata, and then minimizing them, which is known to be undecidable for $\mathbb{T}$-WFAs. Therefore, we adapt a heuristic from Tabakov and Vardi [TV05]: For fixed number of states and alphabet size, we generate transitions in such a way that, for each state q and symbol a, the *expected transition density* — the number of outgoing transitions from q via a with non-$0_{\mathbb{S}}$ weight — is 1.25. For the numeric semirings, these non-$0_{\mathbb{S}}$ weights are sampled uniformly at random from $\{0, \ldots, \min(1000, b)\}$, where b is either the bound for the bounded Tropical semiring or ∞ for the infinite semirings.

6.5.3 RQ1: Ablation Study

We start with our ablation study and investigate:

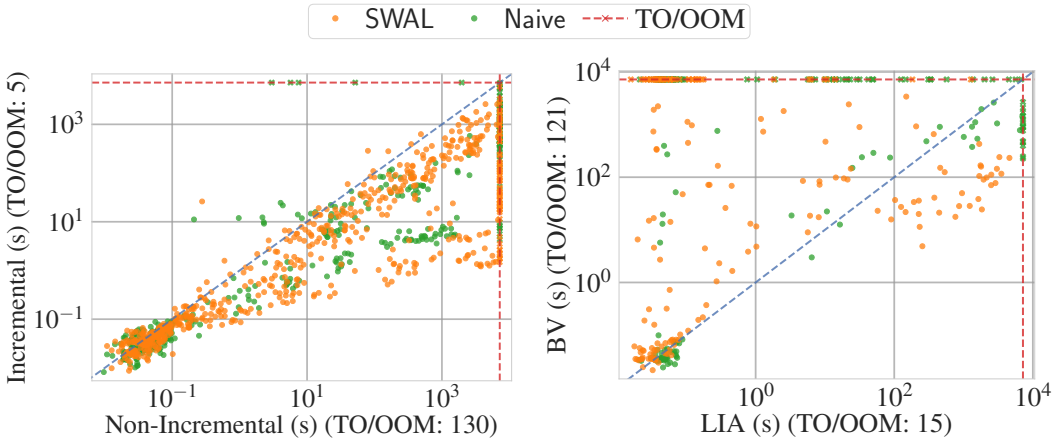
– *Incrementality*: What is the impact of *incremental* SMT solving?

– *Semiring Encoding*: For the bounded Tropical semiring ($b = 100$), what is the impact of the different encodings (QF-BV vs QF-LIA)?

Incrementality Figure 6.2a indicates the *learner time in seconds*, i.e., the total run-time except for computing answers to queries, for both NAIVE and SWAL. We consider *all* benchmarks for all semirings and encodings. Each dot represents one benchmark. The x-axis indicates the learner time *without* incremental SMT solving and the y-axis *with* incremental solving. Whenever a dot is *below* the diagonal line, incremental solving outperforms non-incremental solving (and vice versa). The red dashed line indicates timeouts / out-of-memory faults (TO/OOM).

We observe that incremental solving almost always results in a shorter learning time, with fewer benchmarks hitting TO/OOM (5 for incremental and 130 for non-incremental). This effect is observed in both the NAIVE and SWAL algorithms. We conclude that the incremental nature of the way our constraint sets are built is highly beneficial for incremental SMT solving.

Semiring Encoding Figure 6.2b again indicates the learner time in seconds for all benchmarks over the *bounded* ($b = 100$) Tropical semiring. The x-axis indicates the learner time when using the QF-LIA encoding whereas the y-axis indicates the time for the QF-BV encoding. We investigate both NAIVE and SWAL. Whenever a



(a) Every dot represents a benchmark.

(b) Every dot represents a benchmark.

Figure 6.2: Effect on learner time (seconds in log-log) of incremental SMT solving (left), and bit vector (QF-BV) vs linear integer arithmetic (QF-LIA) encoding for bounded tropical semirings (right). TO/OOM count provided per axis.

dot is *below* the diagonal line, QF-BV outperforms QF-LIA (and vice versa). Even though QF-BV occasionally takes a lead over QF-LIA, the results indicate that QF-LIA mostly outperforms QF-BV. This is particularly striking when considering the number of TO/OOM hits: For QF-BV, 121 benchmarks hit TO/OOM, whereas for QF-LIA, only 15 benchmarks hit these thresholds.

6.5.4 RQ2: Comparison with NAIVE Algorithm

For our comparison of SWAL and NAIVE, we compare (i) how the algorithms scale in terms of state sizes of *learned* automata and (ii) their relative query efficiency, i.e., the number of membership queries required. We consider benchmarks with incrementality, and $|\Sigma| = 2$ for all semirings. Results are depicted in Figure 6.3.

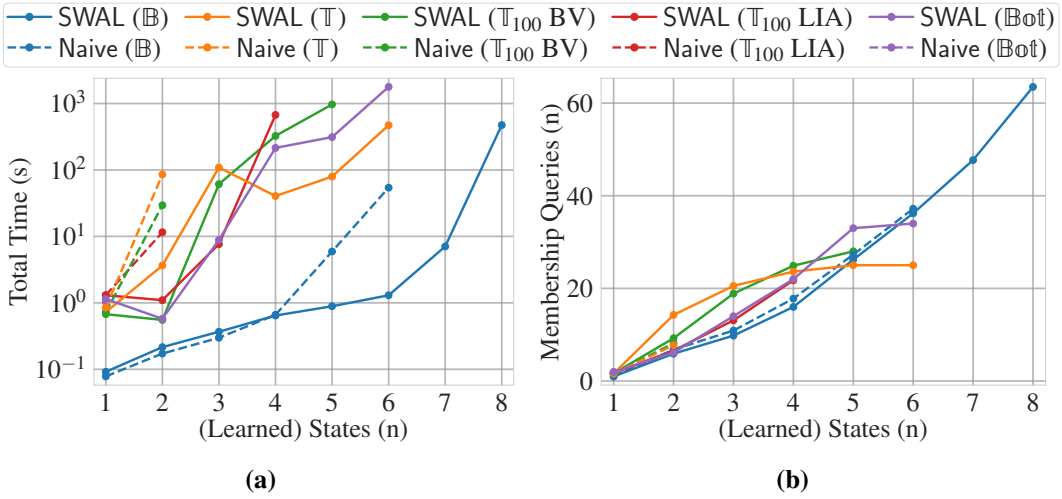


Figure 6.3: Comparison of NAIVE vs SWAL on average total run-time (seconds in log-log, left), and number of membership queries (count, right), over number of states.

Figure 6.3a shows the *average total time* in seconds (log-scale) on the y-axis to learn minimal automata with n-states, where n is indicated on the x-axis. Dashed lines indicate run-times for NAIVE and solid lines run-times for SWAL, each being distinguished by the respective semiring. We observe that SWAL is capable of learning minimal NFAs (i.e., $\mathbb{B}$-WFAs) with up to 8 states within the given time- and memory limits (blue solid line). NAIVE, on the other hand, only learns minimal NFAs with up to 6 states (blue dashed line). For all other semirings, NAIVE even only scales up to at most 2 states, whereas SWAL scales up to 6 states. Moreover, SWAL consistently outperforms NAIVE, sometimes by orders of magnitude. We conclude

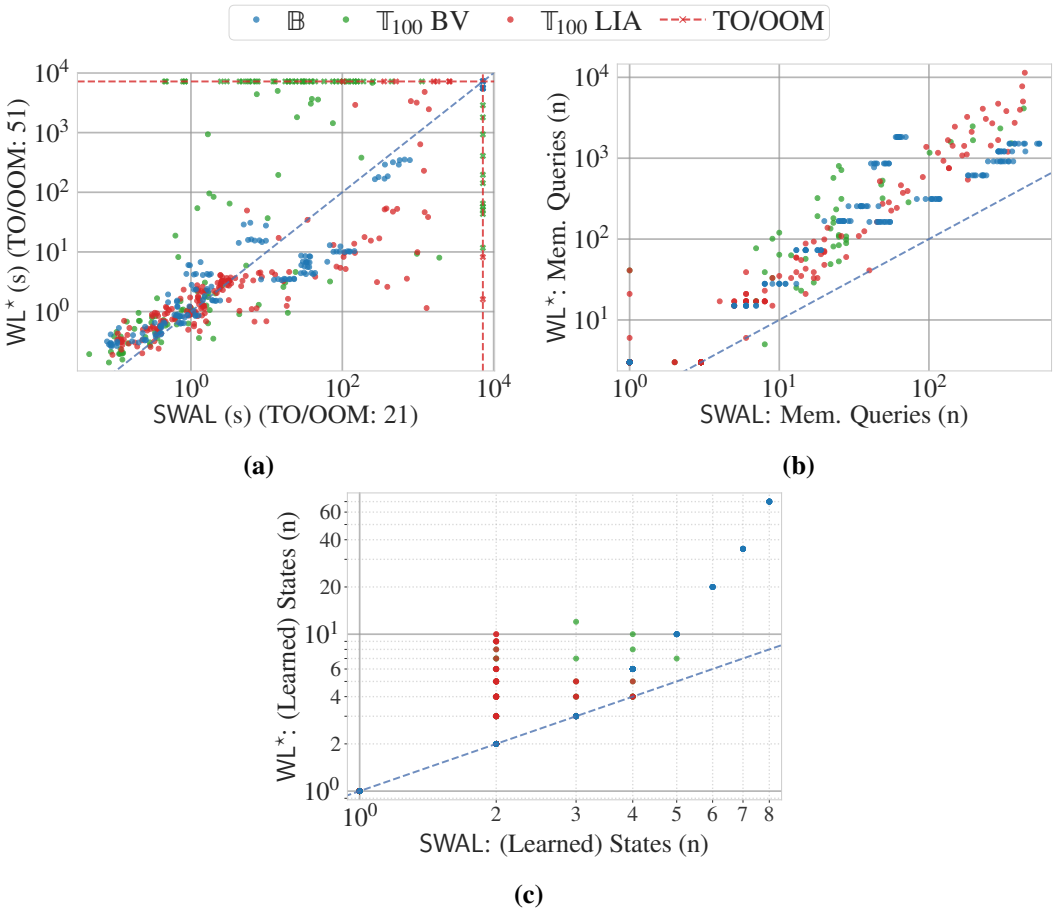


Figure 6.4: Comparison of WL⋆ vs SWAL on total run-time (seconds in log-log, top left), number of membership queries (count in log-log, top right), and number of states learned (log-log, bottom). TO/OOM count provided per axis (top left).

that SWAL is superior to NAIVE in scalability, and that our more sophisticated equation system based on state-languages is highly beneficial for SMT solving.

Figure 6.3b shows the *average number of membership queries* on the y-axis to learn automata with n-states, where n is indicated on the x-axis. As is to be expected, both NAIVE and SWAL require almost the same number of membership queries. Hence, we conclude that SWAL consistently outperforms NAIVE while not requiring more interaction with the teacher.

6.5.5 RQ3: Comparison with WL⋆ Algorithm

WL⋆ [Van+20] is a state-of-the-art L⋆-based active WFA learning algorithm for finite semirings and principal ideal domains. This means, under the semirings

considered in this chapter, WL* can learn WFAs over the Boolean and the *bounded* ($b = 100$) Tropical semiring. In contrast to SWAL, WL* is, however, *not* guaranteed to produce *minimal* automata. We are thus primarily interested in (i) how much the learned automata differ in size and (ii) how many membership queries the respective algorithms require for learning these automata.

Since WL* has not been implemented, we developed our own reference implementation. To enable a fair comparison, we optimize WL* in the same manner as we optimized SWAL, and use the exact same back-end libraries (for SMT solving and matrix operations), semiring encodings, and teachers (with random seeds fixed). Importantly, WL* *always produces automata of the same size, regardless of how it is implemented*, which renders our comparison reasonable.

Results We consider all benchmarks for the Boolean and the bounded ($b = 100$) Tropical semiring. Figure 6.4a indicates the total run-time in seconds for both SWAL (x-axis) and WL* (y-axis) in log-log-scale, distinguished by semiring. Figure 6.4b provides the number of membership queries required by the respective algorithms (log-log scale). Whenever a dot is *above* the diagonal line, SWAL needs *less* membership queries than WL* (and vice versa). Finally, Figure 6.4c gives the state sizes of the *learned* automata for each algorithm. Whenever a dot is *above* the diagonal line, WL* learned *bigger* automata than SWAL.

We observe that SWAL achieves state-minimality *while consistently requiring fewer membership queries* than WL*. The number of states in the learned automata can differ drastically, with automata learned by WL* being up to 10 times bigger than the automata learned by SWAL. In terms of total time, more often than not (Figure 6.4a: 234 vs 163 of 397 terminating points), SWAL terminates faster than WL*, and has a significantly smaller number of TO/OOM hits. We conclude that SWAL consistently learns (sometimes significantly) smaller automata than WL* while requiring fewer membership queries and remaining competitive on run-time.

6.6 Related Work

Active automata learning traces its roots back to the late 1980s with Angluin's seminal $L^\star$ algorithm [Ang87]. While $L^\star$ was tailored for learning deterministic finite automata (DFAs), it is the foundation of a long line of adaptations for learning other types of automata, including (nondeterministic) weighted automata. The first such contribution surfaces in the 1990s by Bergadano and Varricchio [BV96] with an $L^\star$-based algorithm for learning (minimal) weighted automata over fields such as the rationals. In the late 2000s this was further adapted by Bollig et al. [Bol+09] with the $NL^\star$ algorithm to learn nondeterministic finite automata (NFAs), which correspond to weighted automata over the Boolean semiring, however the class of learnable automata was restricted (so-called residual automata) and not necessarily minimal. This procedure was generalized by Van Heerdt et al. [Van+20] with the $WL^\star$ algorithm, which was proved to terminate for all principal ideal domains (yet no minimality guarantee). The recent couple of years have seen a sharp increase in interest in weighted automata learning, with new proofs of termination for the $WL^\star$ algorithm for integer-valued automata over the rationals [Bun+24], automata over number fields with almost minimality [Ari+25], and the subclass of deterministic weighted automata over semifields [Pas+24]. Daviaud and Johnson [DJ25] also provided a more formal language overview of the general feasibility of $L^\star$-style learning of weighted automata, however the complete learnability of further semirings such as the naturals or tropical semirings remains unknown. In the aforementioned works, only the algorithms from Bergadano and Varricchio [BV96], Buna-Marginean et al. [Bun+24], and Aristote et al. [Ari+25] have minimality arguments. The focus is often on using the algebraic properties of the semiring to characterize termination (ignoring minimality) of $WL^\star$ by restricting the semiring or the class of weighted languages the WFA can recognize. Our algorithm instead always guarantees minimality, though this comes at the cost of not having an easy way to explore the same algebraic properties for provable guarantees on termination.

Though we are the first to use SMT for active learning of weighted automata, there is previous work on using satisfiability solvers for inference of deterministic

finite automata from finite positive and negative examples (in a passive learning setting) [BF72; OS01; HV10] and in active learning in the context of an incomplete teacher [Moe+23]. Oliveira and Silva [OS01], extending the approach of Biermann and Feldman [BF72], use constraint solvers to find a mapping from the states of a prefix-tree automaton (PTA, essentially a partial DFA) to an automaton of a fixed size. If the search fails, the target size is increased until an automaton is found. The work of Heule and Verwer [HV10] takes inspiration from graph-coloring to encode a SAT problem to directly infer DFAs from examples. In the active setting, Tappler et al. [TAL22] use SMT encodings to learn timed automata, and Moeller et al. [Moe+23] formulate a SAT encoding to infer missing entries in an observation table for $L^\star$– the motivation being that the teacher might be imperfect or incomplete and not provide answers to all membership queries. All these works are restricted to deterministic automata over the Booleans, whereas our work provides an algorithm in a richer setting of weighted automata over semirings requiring more general theories and solvers.

6.7 Discussion

We have presented a novel SMT-based active automata learning algorithm for WFAs as a practical alternative to Hankel/$L^\star$-style methods. The evaluation of our implementation shows promise: It learns numerous minimal WFAs over both finite and infinite semirings, many of which are out of scope for existing $L^\star$-style algorithms, and is competitive with a state-of-the-art algorithm while producing significantly smaller automata and requiring fewer interactions with the teacher.

As future work, we plan to investigate a more incremental version of our algorithm, where new hypothesis automata are more strongly related to previous ones. Similarly to $L^\star$-based characterizations of termination, this might yield (i) performance increases and (ii) generalizations of Theorem 6.2.5 for termination guarantees for a broader class of semirings. Moreover, we plan to extend our constraint-based approach for learning a broader class of automata including, e.g., probabilistic automata. The existence of active learning for probabilistic languages is

a longstanding open question that is closely related to convex optimization problems and it would be interesting to investigate if an SMT approach would offer interesting trade-offs when compared to other approaches.

Chapter 7

Conclusion and Future Work

In this thesis we have explored how automata learning techniques can be developed for broader applicability in lightweight formal methods, and ultimately for production software systems developed by organizations of all sizes. Across four chapters of contributions, we have addressed obstacles spanning the full pipeline of applied automata learning, from end-to-end tooling to core algorithmic foundations.

After establishing preliminary foundations in Chapter 2, Chapter 3 introduced PROGNOSIS: a modular, end-to-end tool for automated analysis of network protocol implementations, powered by automata learning and program synthesis. Through a variety of analysis methods — including model checking and differential testing — PROGNOSIS was able to detect a wide range of faults in the QUIC protocol: a design flaw in the QUIC specification itself, as well as logic errors, unimplemented behavior, and security vulnerabilities across four globally used implementations.

Despite these findings, the reach of PROGNOSIS was limited by the state of the art in automata learning at the time. Perturbations in the observations of the learning process, caused by, e.g., packet loss over the network, made learning even moderately sized models unfeasible due to the instability of long queries. At the same time, when a flaw was identified in a model, verifying whether a later version of the implementation indeed fixed the issue at hand meant expensively learning a new model from scratch, as the system may have changed in unexpected ways.

To tackle the latter, Chapter 4 presented IKV: an adaptive automata learning algorithm for efficiently updating models based on previously learned behavior. IKV

was the first adaptive algorithm based on KV's classification trees, departing from the observation-table-based adaptive learners that dominated the state of the art at the time, and significantly reducing redundant queries when systems evolve.

Building on this direction, the CƐAL framework of Chapter 5 introduced a practical alternative to the MAT formalism, tackling conflicts in observations — be it from noise or system evolutions — at the framework level. While IKV was a custom adaptation of the existing KV algorithm to the adaptive setting, CƐAL generalizes this idea: any classic MAT learner can be plugged into CƐAL unchanged, and benefit from a learning environment robust to both noise and evolution.

Finally, although PROGNOSIS's models can capture the consequences of logic and control flow errors in a protocol implementation, they lack the expressiveness to capture its *quantitative behavior*. Effects such as latency, reliability, or bandwidth could not be analyzed by PROGNOSIS, as the state of the art in weighted automata learning could not, in practice, handle the semirings that capture such metrics. The SWAL algorithm of Chapter 6 aimed to tackle this, learning *minimal* nondeterministic weighted automata over practical semirings such as the Tropical and Bottleneck, capturing latency and bandwidth, respectively.

While each of the preceding chapters concludes with a discussion of future directions specific to its own contribution, we take here a more global perspective: both on how the contributions of this thesis come together, and on broader directions for automata learning in lightweight formal methods.

Real-World Benchmarking The most immediate direction is to bring IKV, CƐAL, and SWAL into PROGNOSIS itself. While each of these algorithms has been evaluated on synthetic or controlled benchmarks showing their advantage over the state-of-the-art, the original motivation for their development was the practical limitations encountered when applying PROGNOSIS to real-world protocol implementations. Combining them into a single learning pipeline would allow PROGNOSIS to learn larger, more accurate, and quantitative models of network protocol implementations, and we expect this to uncover a new class of bugs that were previously out of reach.

From Protocols to Networks A second direction emerges from PROGNOSIS's position in the networking stack. PROGNOSIS is particularly suited to learning implementations of communication protocols such as TCP, QUIC, SSH, and TLS, but does not address the layer below: the network of routers and forwarding policies on which these protocols rely to ensure deliverability and efficiency across the network. One would like to learn models of this layer directly, rather than only the endpoint implementations that sit on top of it. As mentioned in Section 1.4.5, we have already begun work in this direction, contributing both a learning algorithm for NetKAT [Moe+25] — a language for capturing network behavior grounded in an automata-based semantics — and an extension of NetKAT to the quantitative domain [Suá+26]. A natural next step is to combine these two threads into an active learning algorithm for Weighted NetKAT, enabling automated recovery of quantitative network models directly from interactions with deployed networks. The SWAL algorithm of Chapter 6 provides a starting point for the algorithmic core of such a learner, though significant challenges remain: NetKAT automata rely on symbolic transition systems to efficiently capture the behavior of an entire network over very large packet spaces, and encoding such models into an SMT instance would be considerably more involved than the constraint systems of Chapter 6.

Beyond Myhill–Nerode As discussed in Chapter 2, the Myhill–Nerode theorem is a language-level characterization of the existence of deterministic finite automata, and it underpins the vast majority of active automata learning algorithms. Generalizations of Myhill–Nerode work well for models that retain its core properties — such as symbolic automata [DAn+19] and nominal automata [Moe+17] — but are a poor fit for others: They yield exponential characterizations for timed automata [Wag23], limit the expressiveness of register operations for register automata [Cas+16], and fail to terminate in general for weighted automata [Van+20]. PROGNOSIS and SWAL both sidestep these limitations through SMT solving, recovering both expressiveness and performance. While SMT-based learning has been explored for timed [TAL22] and register automata [SFV18], existing work in this space remains tied to strictly deterministic models and tends to scale suboptimally. It is worth investigating whether

the state-local encodings developed in Chapter 6 can deliver similar performance gains in these settings, and potentially lift the determinism restriction along the way.

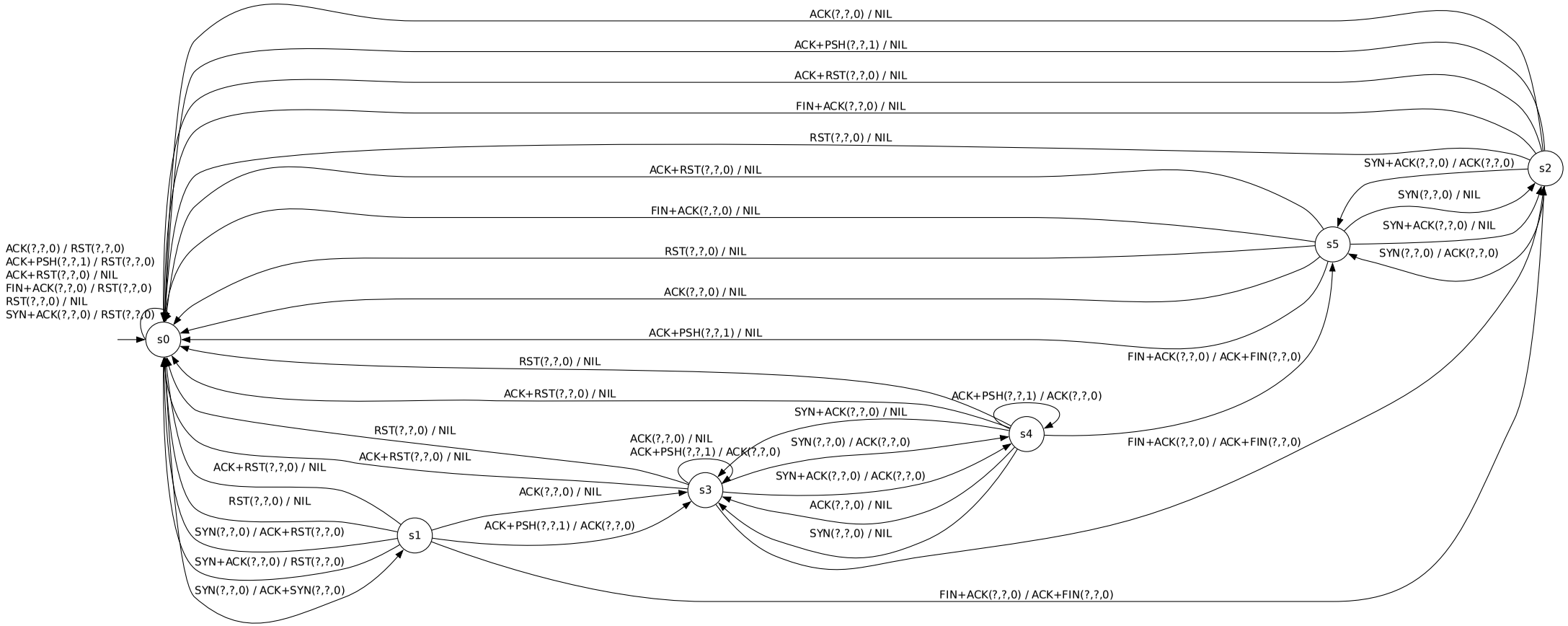
Beyond the MAT Interface On a related note, the MAT interface is itself shaped by its commitment to Myhill–Nerode: Membership and equivalence queries are jointly the minimal interface that suffices to recover the Myhill–Nerode relation of a regular language. Loosening the Myhill–Nerode commitment, as suggested above, also loosens the case for MAT specifically as the canonical learning interface. It is worth asking what other closed-box interfaces might offer comparable theoretical guarantees while being more practical to realize. Recent work has begun exploring this space, with active learning algorithms based on preference queries [HBC25], prefix queries [FRG25], and learning with advice [FO25]. Another direction worth exploring is an oracle that provides general propositional clauses over words, to be fed directly into an SMT encoding such as those of PROGNOSIS and SWAL. Such an interface would sit naturally alongside the SMT-based algorithms developed in this thesis, and could plausibly extend their reach to system models for which neither MAT-style learning nor Myhill–Nerode-based reasoning is currently effective.

Appendix A

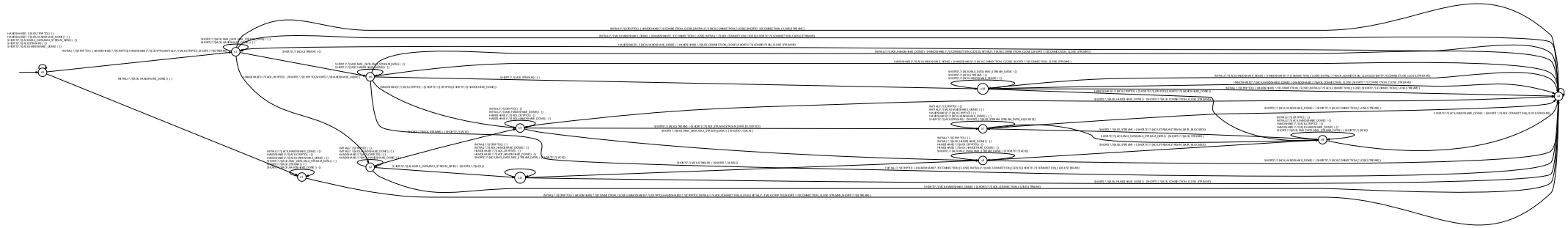
PROGNOSIS Models

A.1 Learned Mealy Machines

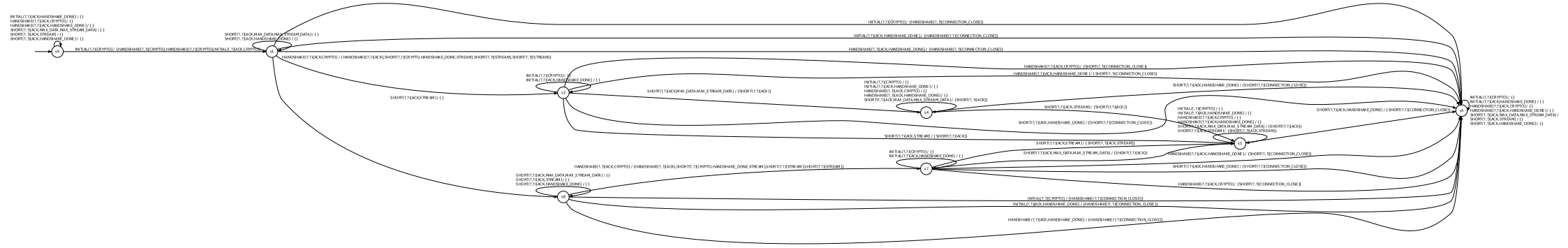
A.1.1 TCP Implementation


ACK(?,?,0) / NIL
ACK+PSH(?,?,1) / NIL
ACK+RST(?,?,0) / NIL
FIN+ACK(?,?,0) / NIL
RST(?,?,0) / NIL
ACK+RST(?,?,0) / NIL
FIN+ACK(?,?,0) / NIL
RST(?,?,0) / NIL
ACK(?,?,0) / NIL
ACK+PSH(?,?,1) / NIL
RST(?,?,0) / NIL
ACK+RST(?,?,0) / NIL
RST(?,?,0) / NIL
ACK+RST(?,?,0) / NIL
RST(?,?,0) / NIL
SYN(?,?,0) / ACK+RST(?,?,0)
SYN+ACK(?,?,0) / RST(?,?,0)
SYN(?,?,0) / ACK+SYN(?,?,0)
ACK(?,?,0) / NIL
ACK+PSH(?,?,1) / ACK(?,?,0)
ACK(?,?,0) / NIL
ACK+PSH(?,?,1) / ACK(?,?,0)
SYN+ACK(?,?,0) / NIL
SYN(?,?,0) / ACK(?,?,0)
SYN+ACK(?,?,0) / ACK(?,?,0)
ACK(?,?,0) / NIL
SYN(?,?,0) / NIL
ACK+PSH(?,?,1) / ACK(?,?,0)
FIN+ACK(?,?,0) / ACK+FIN(?,?,0)
FIN+ACK(?,?,0) / ACK+FIN(?,?,0)
SYN+ACK(?,?,0) / ACK(?,?,0)
SYN(?,?,0) / NIL
SYN+ACK(?,?,0) / NIL
SYN(?,?,0) / ACK(?,?,0)
FIN+ACK(?,?,0) / ACK+FIN(?,?,0)
ACK(?,?,0) / RST(?,?,0)
ACK+PSH(?,?,1) / RST(?,?,0)
ACK+RST(?,?,0) / NIL
FIN+ACK(?,?,0) / RST(?,?,0)
RST(?,?,0) / NIL
SYN+ACK(?,?,0) / RST(?,?,0)
s0
s1
s2
s3
s4
s5

A.1.2 Google's QUIC

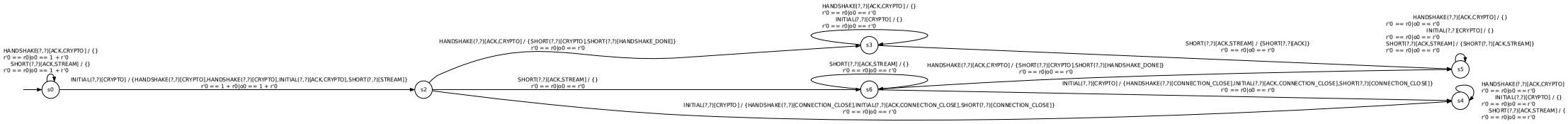


A.1.3 Cloudflare's Quiche


s0
s1
s2
s3
s4
s5
s6
s7
s8
INITIAL(?,?)[CRYPTO]/ {}
INITIAL(?,?)[ACK,HANDSHAKE_DONE] / { }
HANDSHAKE(?,?)[ACK,CRYPTO]/ { }
HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE]/ { }
SHORT(?, ?)[ACK,MAX_DATA,MAX_STREAM_DATA] / { }
SHORT(?, ?)[ACK,STREAM] / { }
SHORT(?, ?)[ACK,HANDSHAKE_DONE] / {}
INITIAL(?,?)[CRYPTO]/ {HANDSHAKE(?, ?)[CRYPTO],HANDSHAKE(?,?)[CRYPTO],INITIAL(?, ?)[ACK,CRYPTO]}
SHORT(?, ?)[ACK,MAX_DATA,MAX_STREAM_DATA] / { }
SHORT(?, ?)[ACK,HANDSHAKE_DONE] / { }
INITIAL(?,?)[CRYPTO]/ {HANDSHAKE(?, ?)[CONNECTION_CLOSE]}
INITIAL(?,?)[ACK,HANDSHAKE_DONE]/ {HANDSHAKE(?,?)[CONNECTION_CLOSE]}
HANDSHAKE(?, ?)[ACK,HANDSHAKE_DONE]/ {HANDSHAKE(?, ?)[CONNECTION_CLOSE]}
HANDSHAKE(?, ?)[ACK,CRYPTO] / { HANDSHAKE(?, ?)[ACK], SHORT(?, ?)[CRYPTO, HANDSHAKE_DONE,STREAM], SHORT(?, ?)[STREAM], SHORT(?, ?)[STREAM]}
HANDSHAKE(?,?)[ACK,CRYPTO]/ {SHORT(?, ?)[CONNECTION_CLOSE]}
HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE]/ { SHORT(?, ?)[CONNECTION_CLOSE]}
INITIAL(?,?)[CRYPTO]/ {}
INITIAL(?,?)[ACK,HANDSHAKE_DONE] / { }
SHORT(?,?)[ACK,MAX_DATA,MAX_STREAM_DATA] / (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,STREAM] / (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,HANDSHAKE_DONE] / (SHORT(?,?)[CONNECTION_CLOSE])
SHORT(?,?)[ACK,STREAM]/ (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,HANDSHAKE_DONE] / (SHORT(?,?)[CONNECTION_CLOSE])
SHORT(?,?)[ACK,STREAM] / (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,STREAM] / (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,STREAM]
SHORT(?,?)[ACK,STREAM] / (SHORT(?,?)[ACK])
SHORT(?,?)[ACK,STREAM] / (SHORT(?,?)[ACK])
INITIAL(?,?)[CRYPTO]/ {}
INITIAL(?,?)[ACK,HANDSHAKE_DONE] / { }
SHORT(?,?)[ACK,MAX_DATA,MAX_STREAM_DATA] / (SHORT(?,?)[ACK])
HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE]/ (SHORT(?, ?)[CONNECTION_CLOSE])
SHORT(?,?)[ACK,HANDSHAKE_DONE] / (SHORT(?,?)[CONNECTION_CLOSE])
HANDSHAKE(?,?)[ACK,CRYPTO]/ (SHORT(?, ?)[CONNECTION_CLOSE])
HANDSHAKE(?, ?)[ACK,CRYPTO] / (HANDSHAKE(?, ?)[ACK], SHORT(?, ?)[CRYPTO, HANDSHAKE_DONE,STREAM],SHORT(?,?)STREAM],SHORT(?,?)STREAM])
SHORT(?,?)[ACK,MAX_DATA,MAX_STREAM_DATA] / {}
SHORT(?,?)[ACK,STREAM] / { }
SHORT(?,?)[ACK,HANDSHAKE_DONE] / { }
INITIALC,?)[CRYPTO]/ {}
INITIAL(?,?)[ACK,HANDSHAKE_DONE] / { }
INITIALC,?)[CRYPTO]/ {HANDSHAKE(?, ?)[CONNECTION_CLOSE]}
INITIALC, ?)[ACK,HANDSHAKE_DONE] / (HANDSHAKE(?, ?)[CONNECTION_CLOSE])
HANDSHAKE(?,?)[ACK,HANDSHAKE_DONE] / (HANDSHAKE(?,?)[CONNECTION_CLOSE])

A.2 Synthesized Extended Mealy Machines

A.2.1 Google's QUIC – `Maximum Stream Data` Field


HANDSHAKE(?,?)[ACK,CRYPTO] / { }
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / { }
r'0 == r0|o0 == r'0
HANDSHAKE(?,?)[ACK,CRYPTO] / { }
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / { }
r'0 == r0|o0 == r'0
SHORT(?,?)[ACK,STREAM] / {SHORT(?,?)[ACK]}
r'0 == r0|o0 == r'0
SHORT(?,?)[ACK,STREAM] / {SHORT(?,?)[ACK,STREAM]}
r'0 == r0|o0 == r'0
HANDSHAKE(?,?)[ACK,CRYPTO] / { }
r'0 == r0|o0 == r'0
SHORT(?,?)[ACK,STREAM] / { }
r'0 == r0|o0 == 1 + r'0
HANDSHAKE(?,?)[ACK,CRYPTO] / {SHORT(?,?)[CRYPTO],SHORT(?,?)[HANDSHAKE_DONE]}
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / {HANDSHAKE(?,?)[CRYPTO],HANDSHAKE(?,?)[CRYPTO],INITIAL(?,?)[ACK,CRYPTO],SHORT(?,?)[STREAM]}
r'0 == 1 + r0|o0 == 1 + r'0
SHORT(?,?)[ACK,STREAM] / { }
r'0 == r0|o0 == r'0
SHORT(?,?)[ACK,STREAM] / { }
r'0 == r0|o0 == r'0
HANDSHAKE(?,?)[ACK,CRYPTO] / {SHORT(?,?)[CRYPTO],SHORT(?,?)[HANDSHAKE_DONE]}
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / {HANDSHAKE(?,?)[CONNECTION_CLOSE],INITIAL(?,?)[ACK,CONNECTION_CLOSE],SHORT(?,?)[CONNECTION_CLOSE]}
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / {HANDSHAKE(?,?)[CONNECTION_CLOSE],INITIAL(?,?)[ACK,CONNECTION_CLOSE],SHORT(?,?)[CONNECTION_CLOSE]}
r'0 == r0|o0 == r'0
HANDSHAKE(?,?)[ACK,CRYPTO]
r'0 == r0|o0 == r'0
INITIAL(?,?)[CRYPTO] / { }
r'0 == r0|o0 == r'0
SHORT(?,?)[ACK,STREAM] / { }
r'0 == r0|o0 == r'0
s0
s2
s3
s4
s5
s6

Appendix B

CƐAL Results

The following table lists the best learning configurations (algorithm and number of repeats) for each framework and each learning experiment (target, type, and level of noise). Each experiment is performed 100 times and times-out after 10 million observations on the SUL. The success rate is the number of successful runs and the observation symbol count is the average number of observation symbols used during a successful learning run.

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
4_learnresult_PIN_fix.dot	MAT	TTT	(5,10)	100	18800.07
INPUT noise at 0.01	CƐAL	L*	(5,10)	100	23667.96
4_learnresult_PIN_fix.dot	MAT	TTT	(5,10)	100	20012.76
OUTPUT noise at 0.01	CƐAL	L*	(5,10)	100	23650.55
4_learnresult_PIN_fix.dot	CƐAL	L*	(5,10)	100	25818.2
INPUT noise at 0.05	MAT	KV	(20,30)	91	106431.74
4_learnresult_PIN_fix.dot	CƐAL	L*	(5,10)	100	25141.6
OUTPUT noise at 0.05	MAT	L*	(20,30)	91	127966.04
4_learnresult_PIN_fix.dot	CƐAL	L*	(10,20)	100	57116.14
INPUT noise at 0.1	MAT	KV	(20,30)	20	53483.85

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
4_learnresult_PIN_fix.dot	CƐAL	L⋆	(5,10)	100	35095.97
OUTPUT noise at 0.1	MAT	TTT	(20,30)	18	36669.28
ASN_learnresult_ MAESTRO_fix.dot	MAT	TTT	(5,10)	100	19454.34
INPUT noise at 0.01	CƐAL	L⋆	(5,10)	100	23669.49
ASN_learnresult_ MAESTRO_fix.dot	MAT	TTT	(5,10)	100	20906.17
OUTPUT noise at 0.01	CƐAL	L⋆	(5,10)	100	23651.8
ASN_learnresult_ MAESTRO_fix.dot	CƐAL	L⋆	(5,10)	100	25683.77
INPUT noise at 0.05	MAT	TTT	(20,30)	91	77650.24
ASN_learnresult_ MAESTRO_fix.dot	CƐAL	L⋆	(5,10)	100	25294.13
OUTPUT noise at 0.05	MAT	L⋆	(20,30)	96	109329.89
ASN_learnresult_ MAESTRO_fix.dot	CƐAL	L⋆	(10,20)	100	56806.28
INPUT noise at 0.1	MAT	L⋆	(20,30)	26	98705.58
ASN_learnresult_ MAESTRO_fix.dot	CƐAL	L⋆	(5,10)	100	33890
OUTPUT noise at 0.1	MAT	KV	(20,30)	19	53137.53
Rabo_learnresult_ MAESTRO_fix.dot	MAT	TTT	(5,10)	100	18143.9
INPUT noise at 0.01	CƐAL	L⋆	(5,10)	100	23661.77

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
Rabo_learnresult_ MAESTRO_fix.dot	MAT	TTT	(5,10)	100	20639.11
OUTPUT noise at 0.01	CƐAL	L*	(5,10)	100	23652.36
Rabo_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	25762.24
INPUT noise at 0.05	MAT	TTT	(20,30)	93	78030.14
Rabo_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	25272.71
OUTPUT noise at 0.05	MAT	L*	(20,30)	93	155886.7
Rabo_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	56333.1
INPUT noise at 0.1	MAT	L*	(20,30)	25	65872.72
Rabo_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	34832.46
OUTPUT noise at 0.1	MAT	L*	(20,30)	24	79005.62
Rabo_learnresult_ SecureCode_Aut_fix.dot	MAT	TTT	(5,10)	100	13191.67
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	28092.53
Rabo_learnresult_ SecureCode_Aut_fix.dot	MAT	TTT	(5,10)	100	11904.83
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	27709.41
Rabo_learnresult_ SecureCode_Aut_fix.dot	CƐAL	L*	(5,10)	100	31835.22

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.05	MAT	KV	(20,30)	95	80110.18
Rabo_learnresult_ SecureCode_Aut_fix.dot	CƐAL	L⋆	(5,10)	100	30321.21
OUTPUT noise at 0.05	MAT	KV	(20,30)	94	81503.62
Rabo_learnresult_ SecureCode_Aut_fix.dot	CƐAL	L⋆	(10,20)	100	71161.74
INPUT noise at 0.1	MAT	L⋆	(20,30)	40	68508.25
Rabo_learnresult_ SecureCode_Aut_fix.dot	CƐAL	L⋆	(5,10)	100	43735.72
OUTPUT noise at 0.1	MAT	L⋆	(20,30)	34	68287.74
GnuTLS_3.3.12_client_ regular.dot	CƐAL	L⋆	(5,10)	100	8700.15
INPUT noise at 0.01	MAT	KV	(5,10)	100	13312.08
GnuTLS_3.3.12_client_ regular.dot	CƐAL	L⋆	(5,10)	100	8714.03
OUTPUT noise at 0.01	MAT	KV	(5,10)	100	14611.65
GnuTLS_3.3.12_client_ regular.dot	CƐAL	L⋆	(5,10)	100	9383.59
INPUT noise at 0.05	MAT	KV	(5,10)	100	13765.59
GnuTLS_3.3.12_client_ regular.dot	CƐAL	L⋆	(5,10)	100	9542.08
OUTPUT noise at 0.05	MAT	KV	(10,20)	100	31961.76

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
GnuTLS_3.3.12_client_regular.dot	CℰAL	L⋆	(5,10)	100	16366.54
INPUT noise at 0.1	MAT	KV	(20,30)	100	56634.7
GnuTLS_3.3.12_client_regular.dot	CℰAL	L⋆	(5,10)	100	13181.22
OUTPUT noise at 0.1	MAT	L⋆	(20,30)	99	108314.29
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	8700.66
INPUT noise at 0.01	MAT	KV	(5,10)	100	12742.74
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	8716.5
OUTPUT noise at 0.01	MAT	KV	(5,10)	100	12744.21
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	9238.11
INPUT noise at 0.05	MAT	L⋆	(5,10)	100	21829.48
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	9579.21
OUTPUT noise at 0.05	MAT	KV	(10,20)	100	23058.97
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	16750.42
INPUT noise at 0.1	MAT	KV	(20,30)	100	52193.66
GnuTLS_3.3.12_server_regular.dot	CℰAL	L⋆	(5,10)	100	13136.49

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OUTPUT noise at 0.1	MAT	$L^\star$	(20,30)	100	109055.26
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(5,10)	100	9678.38
INPUT noise at 0.01	MAT	KV	(5,10)	100	12744.42
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(5,10)	100	9684.25
OUTPUT noise at 0.01	MAT	TTT	(5,10)	100	12932.33
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(5,10)	100	10643.94
INPUT noise at 0.05	MAT	TTT	(5,10)	100	14856.42
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(5,10)	100	10755.87
OUTPUT noise at 0.05	MAT	TTT	(5,10)	100	14001.56
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(10,20)	100	24055.3
INPUT noise at 0.1	MAT	KV	(10,20)	100	27046.64
NSS_3.17.4_client_ regular.dot	CƐAL	$L^\star$	(5,10)	100	16435.53
OUTPUT noise at 0.1	MAT	TTT	(10,20)	100	31956.55
OpenSSL_1.0.2_server_ regular.dot	CƐAL	$L^\star$	(5,10)	100	6417.43
INPUT noise at 0.01	MAT	KV	(5,10)	100	10038.55

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OpenSSL_1.0.2_server_regular.dot	CƐAL	L⋆	(5,10)	100	6434.04
OUTPUT noise at 0.01	MAT	KV	(5,10)	100	9668.83
OpenSSL_1.0.2_server_regular.dot	CƐAL	L⋆	(5,10)	100	6825.86
INPUT noise at 0.05	MAT	KV	(10,20)	100	19902.09
OpenSSL_1.0.2_server_regular.dot	CƐAL	L⋆	(5,10)	100	7066.48
OUTPUT noise at 0.05	MAT	KV	(5,10)	100	10986.84
OpenSSL_1.0.2_server_regular.dot	CƐAL	L⋆	(5,10)	100	10072.76
INPUT noise at 0.1	MAT	TTT	(10,20)	100	24247.78
OpenSSL_1.0.2_server_regular.dot	CƐAL	L⋆	(5,10)	100	8997.39
OUTPUT noise at 0.1	MAT	KV	(20,30)	100	51616.12
Volksbank_learnresult_MAESTRO_fix.dot	MAT	TTT	(5,10)	100	7943.4
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	17237.49
Volksbank_learnresult_MAESTRO_fix.dot	MAT	TTT	(5,10)	100	8030.57
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	17367.51
Volksbank_learnresult_MAESTRO_fix.dot	CƐAL	L⋆	(5,10)	100	34281.38

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.05	MAT	L*	(20,30)	99	63789.12
Volksbank_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	32941.97
OUTPUT noise at 0.05	MAT	L*	(20,30)	100	56836.2
Volksbank_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(10,20)	100	80374.4
INPUT noise at 0.1	MAT	L*	(20,30)	68	55511.59
Volksbank_learnresult_ MAESTRO_fix.dot	CƐAL	L*	(5,10)	100	51674.12
OUTPUT noise at 0.1	MAT	L*	(20,30)	71	55008.2
NSS_3.17.4_server_ regular.dot	CƐAL	L*	(5,10)	100	10638.69
INPUT noise at 0.01	MAT	TTT	(5,10)	100	12000.78
NSS_3.17.4_server_ regular.dot	CƐAL	L*	(5,10)	100	10647.57
OUTPUT noise at 0.01	MAT	TTT	(5,10)	100	11445.93
NSS_3.17.4_server_ regular.dot	CƐAL	L*	(5,10)	100	11705.27
INPUT noise at 0.05	MAT	TTT	(10,20)	100	26472.28
NSS_3.17.4_server_ regular.dot	CƐAL	L*	(5,10)	100	11767.46
OUTPUT noise at 0.05	MAT	TTT	(5,10)	100	12457.79

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
NSS_3.17.4_server_regular.dot	CƐAL	L⋆	(10,20)	100	25801.2
INPUT noise at 0.1	MAT	KV	(10,20)	100	31439.29
NSS_3.17.4_server_regular.dot	CƐAL	L⋆	(5,10)	100	17591.27
OUTPUT noise at 0.1	MAT	TTT	(20,30)	100	59005.09
GnuTLS_3.3.12_client_full.dot	MAT	TTT	(5,10)	100	91160.43
INPUT noise at 0.01	CƐAL	KV	(5,10)	100	205658.94
GnuTLS_3.3.12_client_full.dot	MAT	TTT	(5,10)	100	113987.9
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	194448.25
GnuTLS_3.3.12_client_full.dot	CƐAL	TTT	(5,10)	100	201814.9
INPUT noise at 0.05	MAT	TTT	(10,20)	100	227056.17
GnuTLS_3.3.12_client_full.dot	CƐAL	TTT	(5,10)	100	228027.96
OUTPUT noise at 0.05	MAT	KV	(10,20)	100	247702.66
GnuTLS_3.3.12_client_full.dot	MAT	KV	(20,30)	100	415169.16
INPUT noise at 0.1	CƐAL	TTT	(10,20)	100	466326.9
GnuTLS_3.3.12_client_full.dot	CƐAL	TTT	(10,20)	100	550787.09

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OUTPUT noise at 0.1	MAT	KV	(20,30)	98	555802.05
GnuTLS_3.3.12_server_full.dot	MAT	KV	(5,10)	100	36121.47
INPUT noise at 0.01	CℰAL	TTT	(5,10)	100	73974.93
GnuTLS_3.3.12_server_full.dot	MAT	KV	(5,10)	100	33031.4
OUTPUT noise at 0.01	CℰAL	TTT	(5,10)	100	54344.32
GnuTLS_3.3.12_server_full.dot	CℰAL	KV	(5,10)	100	61347.94
INPUT noise at 0.05	MAT	TTT	(10,20)	100	71734.43
GnuTLS_3.3.12_server_full.dot	CℰAL	$L^\sharp$	(5,10)	100	64307.09
OUTPUT noise at 0.05	MAT	TTT	(10,20)	100	68059.76
GnuTLS_3.3.12_server_full.dot	CℰAL	TTT	(5,10)	100	64513.07
INPUT noise at 0.1	MAT	TTT	(20,30)	100	138291
GnuTLS_3.3.12_server_full.dot	CℰAL	$L^\sharp$	(5,10)	100	48661.21
OUTPUT noise at 0.1	MAT	$L^\sharp$	(20,30)	100	178243.28
learnresult_fix.dot	MAT	TTT	(5,10)	100	13561.85
INPUT noise at 0.01	CℰAL	TTT	(5,10)	100	25606.65
learnresult_fix.dot	MAT	TTT	(5,10)	100	13682.06
OUTPUT noise at 0.01	CℰAL	TTT	(5,10)	100	25727.23

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
learnresult_fix.dot	CƐAL	TTT	(5,10)	100	54883.73
INPUT noise at 0.05	MAT	TTT	(20,30)	100	57206.25
learnresult_fix.dot	CƐAL	L*	(5,10)	100	50620.84
OUTPUT noise at 0.05	MAT	L*	(10,20)	99	49926.49
learnresult_fix.dot	CƐAL	L*	(10,20)	100	121189.06
INPUT noise at 0.1	MAT	L*	(20,30)	48	82305.83
learnresult_fix.dot	CƐAL	L*	(5,10)	100	85413.46
OUTPUT noise at 0.1	MAT	L*	(20,30)	74	94502.73
OpenSSL_1.0.2_client_full.dot	MAT	TTT	(5,10)	100	68973.07
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	135919.55
OpenSSL_1.0.2_client_full.dot	MAT	TTT	(5,10)	100	65188.04
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	117615.6
OpenSSL_1.0.2_client_full.dot	CƐAL	TTT	(5,10)	100	107252.09
INPUT noise at 0.05	MAT	KV	(10,20)	100	130950.71
OpenSSL_1.0.2_client_full.dot	MAT	KV	(10,20)	100	141440.98
OUTPUT noise at 0.05	CƐAL	KV	(5,10)	100	149671.51
OpenSSL_1.0.2_client_full.dot	CƐAL	KV	(10,20)	100	219337.62

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.1	MAT	TTT	(20,30)	100	301707.18
OpenSSL_1.0.2_client_full.dot	MAT	KV	(20,30)	100	298232.72
OUTPUT noise at 0.1	CƐAL	TTT	(10,20)	100	415644.59
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	11271.48
INPUT noise at 0.01	MAT	TTT	(5,10)	100	14535.15
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	11274.89
OUTPUT noise at 0.01	MAT	TTT	(5,10)	100	15962.98
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	11994.71
INPUT noise at 0.05	MAT	TTT	(10,20)	100	34042.37
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	12175.24
OUTPUT noise at 0.05	MAT	TTT	(10,20)	100	35512.27
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	20307.92
INPUT noise at 0.1	MAT	TTT	(20,30)	100	59977.7
RSA_BSAFE_C_4.0.4_server_regular.dot	CƐAL	L*	(5,10)	100	15649.11
OUTPUT noise at 0.1	MAT	TTT	(10,20)	100	38083.16

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	11092.08
INPUT noise at 0.01	MAT	TTT	(5,10)	100	17154.8
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	11104.54
OUTPUT noise at 0.01	MAT	TTT	(5,10)	100	16591.06
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	12026.39
INPUT noise at 0.05	MAT	TTT	(10,20)	100	38671.17
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	12157.83
OUTPUT noise at 0.05	MAT	TTT	(10,20)	100	42172.35
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	22434.34
INPUT noise at 0.1	MAT	TTT	(20,30)	100	81388.63
OpenSSL_1.0.1g_client_regular.dot	CℰAL	L⋆	(5,10)	100	16134.44
OUTPUT noise at 0.1	MAT	TTT	(20,30)	99	93221.85
OpenSSL_1.0.1l_server_regular.dot	MAT	KV	(5,10)	100	83568.33
INPUT noise at 0.01	CℰAL	KV	(5,10)	100	178480.85
OpenSSL_1.0.1l_server_regular.dot	MAT	KV	(5,10)	100	76702.16

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	170014.66
OpenSSL_1.0.1l_server_regular.dot	CƐAL	KV	(5,10)	100	168956.14
INPUT noise at 0.05	MAT	TTT	(10,20)	100	219047.93
OpenSSL_1.0.1l_server_regular.dot	MAT	KV	(10,20)	100	166207.58
OUTPUT noise at 0.05	CƐAL	KV	(5,10)	100	201153.28
OpenSSL_1.0.1l_server_regular.dot	MAT	KV	(20,30)	100	400721.07
INPUT noise at 0.1	CƐAL	KV	(10,20)	100	416222.86
OpenSSL_1.0.1l_server_regular.dot	CƐAL	KV	(10,20)	100	662679.52
OUTPUT noise at 0.1	MAT	KV	(20,30)	94	467222.21
GnuTLS_3.3.8_client_regular.dot	MAT	KV	(5,10)	100	1074206.02
INPUT noise at 0.01	CƐAL	KV	(5,10)	100	2270972.48
GnuTLS_3.3.8_client_regular.dot	MAT	KV	(5,10)	100	1157429.8
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	2363794.93
GnuTLS_3.3.8_client_regular.dot	CƐAL	KV	(5,10)	100	1547844.22
INPUT noise at 0.05	MAT	TTT	(10,20)	100	3137292.43

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
GnuTLS_3.3.8_client_regular.dot	CƐAL	KV	(5,10)	100	2251216.77
OUTPUT noise at 0.05	MAT	L$^\sharp$	(20,30)	100	5885067.18
GnuTLS_3.3.8_client_regular.dot	CƐAL	TTT	(10,20)	100	6898721.62
INPUT noise at 0.1	MAT	KV	(20,30)	88	4581787.99
GnuTLS_3.3.8_client_regular.dot	CƐAL	KV	(20,30)	100	7484163.22
OUTPUT noise at 0.1	MAT	KV	(20,30)	18	1744942.17
NSS_3.17.4_client_full.dot	MAT	KV	(5,10)	100	151067.11
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	254432.67
NSS_3.17.4_client_full.dot	MAT	L$^\sharp$	(5,10)	100	120249.1
OUTPUT noise at 0.01	CƐAL	L$^\sharp$	(5,10)	100	205706.28
NSS_3.17.4_client_full.dot	CƐAL	TTT	(5,10)	100	212075.56
INPUT noise at 0.05	MAT	TTT	(10,20)	100	220689.75
NSS_3.17.4_client_full.dot	CƐAL	TTT	(5,10)	100	155770.93
OUTPUT noise at 0.05	MAT	TTT	(10,20)	100	215655.37
NSS_3.17.4_client_full.dot	MAT	TTT	(20,30)	100	502452.08
INPUT noise at 0.1	CƐAL	TTT	(10,20)	100	532995.13
NSS_3.17.4_client_full.dot	CƐAL	KV	(5,10)	100	277081.72
OUTPUT noise at 0.1	MAT	TTT	(20,30)	97	542911.65

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OpenSSL_1.0.1j_server_regular.dot	MAT	KV	(5,10)	100	98860.19
INPUT noise at 0.01	CϵAL	TTT	(5,10)	100	187712.24
OpenSSL_1.0.1j_server_regular.dot	MAT	KV	(5,10)	100	114944.42
OUTPUT noise at 0.01	CϵAL	TTT	(5,10)	100	192309.94
OpenSSL_1.0.1j_server_regular.dot	CϵAL	KV	(5,10)	100	193923.29
INPUT noise at 0.05	MAT	KV	(10,20)	100	195054.36
OpenSSL_1.0.1j_server_regular.dot	MAT	KV	(10,20)	100	232477.71
OUTPUT noise at 0.05	CϵAL	KV	(5,10)	100	234364.67
OpenSSL_1.0.1j_server_regular.dot	MAT	KV	(20,30)	100	427809.6
INPUT noise at 0.1	CϵAL	$L^\sharp$	(10,20)	100	488676.44
OpenSSL_1.0.1j_server_regular.dot	CϵAL	KV	(10,20)	100	779067.68
OUTPUT noise at 0.1	MAT	TTT	(20,30)	83	501805.19
GnuTLS_3.3.8_server_regular.dot	MAT	$L^\star$	(5,10)	100	305769.92
INPUT noise at 0.01	CϵAL	$L^\star$	(5,10)	100	583269.31
GnuTLS_3.3.8_server_regular.dot	MAT	$L^\star$	(5,10)	100	317981.5

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OUTPUT noise at 0.01	CƐAL	L*	(5,10)	100	565932.75
GnuTLS_3.3.8_server_regular.dot	MAT	L*	(10,20)	100	593180.14
INPUT noise at 0.05	CƐAL	KV	(5,10)	100	712136.47
GnuTLS_3.3.8_server_regular.dot	CƐAL	KV	(5,10)	100	978550.49
OUTPUT noise at 0.05	MAT	L$^\sharp$	(10,20)	100	1124654.69
GnuTLS_3.3.8_server_regular.dot	CƐAL	KV	(10,20)	100	1526791.66
INPUT noise at 0.1	MAT	L*	(20,30)	98	1425113.07
GnuTLS_3.3.8_server_regular.dot	CƐAL	KV	(10,20)	100	2954613.67
OUTPUT noise at 0.1	MAT	L*	(20,30)	61	1067245.87
TCP_FreeBSD_Client.dot	MAT	TTT	(5,10)	100	51471.43
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	83117.33
TCP_FreeBSD_Client.dot	MAT	TTT	(5,10)	100	49144.13
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	95920.54
TCP_FreeBSD_Client.dot	CƐAL	KV	(10,20)	100	263038.67
INPUT noise at 0.05	MAT	L*	(20,30)	87	279107.21
TCP_FreeBSD_Client.dot	CƐAL	KV	(10,20)	100	344807.82
OUTPUT noise at 0.05	MAT	L*	(20,30)	86	248240.19
TCP_FreeBSD_Client.dot	CƐAL	TTT	(20,30)	100	4443575.83

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.1	MAT	L*	(20,30)	22	149063.14
TCP_FreeBSD_Client.dot	CƐAL	TTT	(20,30)	100	6302307.48
OUTPUT noise at 0.1	MAT	L*	(20,30)	11	135983.09
TCP_Windows8_Client.dot	MAT	TTT	(5,10)	100	48855.27
INPUT noise at 0.01	CƐAL	L$^\sharp$	(5,10)	100	78782.79
TCP_Windows8_Client.dot	MAT	TTT	(5,10)	100	48247
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	90816.92
TCP_Windows8_Client.dot	CƐAL	KV	(5,10)	100	95279.01
INPUT noise at 0.05	MAT	L*	(10,20)	100	157329.34
TCP_Windows8_Client.dot	CƐAL	TTT	(5,10)	100	94385.49
OUTPUT noise at 0.05	MAT	KV	(10,20)	100	109640.61
TCP_Windows8_Client.dot	CƐAL	TTT	(10,20)	100	241756.52
INPUT noise at 0.1	MAT	TTT	(20,30)	93	214732.66
TCP_Windows8_Client.dot	CƐAL	TTT	(10,20)	100	236025.16
OUTPUT noise at 0.1	MAT	L*	(20,30)	93	319211.55
GnuTLS_3.3.8_client_full.dot	MAT	KV	(5,10)	100	6374596.78
INPUT noise at 0.01	CƐAL	KV	(5,10)	100	9383499.02
GnuTLS_3.3.8_client_full.dot	MAT	TTT	(5,10)	100	7192497.09
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	10209832.74
GnuTLS_3.3.8_client_full.dot	CƐAL	TTT	(10,20)	100	20145049
INPUT noise at 0.05	MAT	KV	(20,30)	100	23124047.98

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
GnuTLS_3.3.8_client_full.dot	CƐAL	L*	(10,20)	100	22228894.2
OUTPUT noise at 0.05	MAT	L*	(20,30)	100	26922957.92
GnuTLS_3.3.8_client_full.dot	CƐAL	L*	(20,30)	100	42055100.5
INPUT noise at 0.1	MAT	L*	(20,30)	56	21071401.11
GnuTLS_3.3.8_client_full.dot	CƐAL	TTT	(5,10)	99	19156049.74
OUTPUT noise at 0.1	MAT	L$^\sharp$	(20,30)	3	4953119
TCP_Linux_Client.dot	MAT	TTT	(5,10)	100	215066.05
INPUT noise at 0.01	CƐAL	L$^\sharp$	(5,10)	100	325839.84
TCP_Linux_Client.dot	MAT	TTT	(5,10)	100	183551.34
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	337599.48
TCP_Linux_Client.dot	CƐAL	TTT	(10,20)	100	1689401.93
INPUT noise at 0.05	MAT	L*	(20,30)	51	802177.67
TCP_Linux_Client.dot	CƐAL	KV	(20,30)	100	1872055.85
OUTPUT noise at 0.05	MAT	L*	(20,30)	49	801882.9
TCP_Linux_Client.dot	CƐAL	TTT	(20,30)	99	15218384.08
INPUT noise at 0.1	MAT	TTT	(20,30)	1	105619
TCP_Linux_Client.dot	CƐAL	TTT	(10,20)	95	24065009.26
OUTPUT noise at 0.1	MAT	TTT	(10,20)	1	57169
GnuTLS_3.3.8_server_full.dot	MAT	L*	(5,10)	100	1257996.28
INPUT noise at 0.01	CƐAL	L*	(5,10)	100	2066983.82

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
GnuTLS_3.3.8_server_full.dot	MAT	KV	(5,10)	100	1425157.15
OUTPUT noise at 0.01	CƐAL	L*	(5,10)	100	1850643.2
GnuTLS_3.3.8_server_full.dot	CƐAL	L$^\sharp$	(5,10)	100	1992378.22
INPUT noise at 0.05	MAT	L*	(10,20)	100	2695293.81
GnuTLS_3.3.8_server_full.dot	CƐAL	KV	(5,10)	100	3939240.45
OUTPUT noise at 0.05	MAT	KV	(20,30)	100	6560567.49
GnuTLS_3.3.8_server_full.dot	CƐAL	L$^\sharp$	(10,20)	100	6213300.73
INPUT noise at 0.1	MAT	L*	(20,30)	97	5499339.33
GnuTLS_3.3.8_server_full.dot	CƐAL	KV	(10,20)	100	10583663.38
OUTPUT noise at 0.1	MAT	L*	(20,30)	22	4574410
OpenSSL_1.0.1g_server_regular.dot	MAT	KV	(5,10)	100	151437.67
INPUT noise at 0.01	CƐAL	KV	(5,10)	100	278231.68
OpenSSL_1.0.1g_server_regular.dot	MAT	KV	(5,10)	100	124770.08
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	239356.33
OpenSSL_1.0.1g_server_regular.dot	CƐAL	L$^\sharp$	(5,10)	100	219344.8

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.05	MAT	KV	(10,20)	100	313288.07
OpenSSL_1.0.1g_server_regular.dot	CℰAL	KV	(5,10)	100	227970.48
OUTPUT noise at 0.05	MAT	KV	(10,20)	100	291552.03
OpenSSL_1.0.1g_server_regular.dot	CℰAL	KV	(10,20)	100	609691.3
INPUT noise at 0.1	MAT	KV	(20,30)	100	749551.04
OpenSSL_1.0.1g_server_regular.dot	CℰAL	KV	(10,20)	100	756468.39
OUTPUT noise at 0.1	MAT	KV	(20,30)	77	560775.84
DropBear.dot	MAT	$L^\star$	(5,10)	100	137649.68
INPUT noise at 0.01	CℰAL	$L^\sharp$	(5,10)	100	191024.45
DropBear.dot	MAT	$L^\sharp$	(5,10)	100	118485.35
OUTPUT noise at 0.01	CℰAL	$L^\sharp$	(5,10)	100	172474.12
DropBear.dot	CℰAL	TTT	(5,10)	100	657186.2
INPUT noise at 0.05	MAT	$L^\star$	(20,30)	96	657705.43
DropBear.dot	CℰAL	TTT	(10,20)	100	746915.07
OUTPUT noise at 0.05	MAT	TTT	(20,30)	91	749198.88
DropBear.dot	CℰAL	TTT	(20,30)	100	3770482.81
INPUT noise at 0.1	MAT	$L^\star$	(20,30)	2	333079.5
DropBear.dot	CℰAL	TTT	(20,30)	100	3992875.08
OUTPUT noise at 0.1	MAT	KV	(20,30)	2	247372

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
OpenSSH.dot	CƐAL	KV	(5,10)	100	4955202.9
INPUT noise at 0.01	MAT	KV	(10,20)	100	5742879.77
OpenSSH.dot	MAT	L*	(10,20)	100	5354862.63
OUTPUT noise at 0.01	CƐAL	KV	(5,10)	100	5865961.19
OpenSSH.dot	CƐAL	KV	(20,30)	100	19169545.12
INPUT noise at 0.05	MAT	L*	(20,30)	17	3483515.06
OpenSSH.dot	CƐAL	KV	(20,30)	100	23365791.81
OUTPUT noise at 0.05	MAT	L*	(20,30)	8	3307257.12
model4.dot	MAT	TTT	(5,10)	100	196191.43
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	274268.45
model4.dot	MAT	TTT	(5,10)	100	202228.35
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	252335.04
model4.dot	CƐAL	TTT	(10,20)	100	908233.13
INPUT noise at 0.05	MAT	KV	(20,30)	100	1196013.58
model4.dot	CƐAL	TTT	(10,20)	100	596577.7
OUTPUT noise at 0.05	MAT	TTT	(20,30)	100	880001.68
model4.dot	CƐAL	TTT	(20,30)	100	6163432.08
INPUT noise at 0.1	MAT	L*	(20,30)	29	2072063.1
model4.dot	CƐAL	TTT	(20,30)	100	2565861.45
OUTPUT noise at 0.1	MAT	TTT	(20,30)	47	932215.09
model1.dot	MAT	TTT	(5,10)	100	108632.49

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	145286.78
model1.dot	MAT	TTT	(5,10)	100	110363.06
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	157779.41
model1.dot	CƐAL	TTT	(10,20)	100	457462.53
INPUT noise at 0.05	MAT	TTT	(20,30)	98	537245.61
model1.dot	CƐAL	TTT	(10,20)	100	852299.13
OUTPUT noise at 0.05	MAT	L*	(20,30)	76	745257.8
model1.dot	CƐAL	TTT	(20,30)	100	3020350.5
INPUT noise at 0.1	MAT	L*	(20,30)	15	723103.27
model1.dot	CƐAL	TTT	(20,30)	92	12212079.93
OUTPUT noise at 0.1	MAT	L*	(20,30)	1	421758
TCP_Windows8_Server.dot	MAT	L*	(10,20)	100	1958923.06
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	3732066.25
TCP_Windows8_Server.dot	MAT	L*	(5,10)	100	905053.04
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	3473018.13
TCP_Windows8_Server.dot	CƐAL	TTT	(10,20)	100	11418676.36
INPUT noise at 0.05	MAT	L*	(20,30)	96	4012156.69
TCP_Windows8_Server.dot	MAT	L*	(20,30)	100	4080149.32
OUTPUT noise at 0.05	CƐAL	KV	(10,20)	100	7948493.19
TCP_Windows8_Server.dot	CƐAL	TTT	(20,30)	100	25669726.36
OUTPUT noise at 0.1	MAT	TTT	(20,30)	7	3338306.29

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
TCP_FreeBSD_Server.dot	CƐAL	TTT	(5,10)	100	3139842.78
INPUT noise at 0.01	MAT	L*	(10,20)	100	4470383.1
TCP_FreeBSD_Server.dot	CƐAL	TTT	(5,10)	100	3178107.17
OUTPUT noise at 0.01	MAT	TTT	(10,20)	100	4242473.71
TCP_FreeBSD_Server.dot INPUT noise at 0.05	CƐAL	TTT	(20,30)	100	19444312.79
TCP_FreeBSD_Server.dot	CƐAL	TTT	(20,30)	100	19933573.85
OUTPUT noise at 0.05	MAT	L*	(20,30)	4	5323173.75
TCP_Linux_Server.dot	MAT	L*	(10,20)	100	3473612.19
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	4344517.52
TCP_Linux_Server.dot	MAT	L*	(10,20)	100	3551066.4
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	3687906.85
TCP_Linux_Server.dot	CƐAL	TTT	(20,30)	98	40306528.1
INPUT noise at 0.05	MAT	L*	(20,30)	4	5661189
TCP_Linux_Server.dot	CƐAL	TTT	(20,30)	99	31894250.28
OUTPUT noise at 0.05	MAT	L*	(20,30)	7	5675121.57
model3.dot	MAT	TTT	(5,10)	100	376951.11
INPUT noise at 0.01	CƐAL	TTT	(5,10)	100	478557.28
model3.dot	MAT	TTT	(5,10)	100	352924.64
OUTPUT noise at 0.01	CƐAL	TTT	(5,10)	100	460747.47
model3.dot	CƐAL	TTT	(10,20)	100	1502765.59
INPUT noise at 0.05	MAT	KV	(20,30)	99	2196917.95

Table B.1 continued from previous page

Experiment	Fwork.	Algo.	Sample	Succ.	Symbols
model3.dot	MAT	TTT	(10,20)	100	794580.92
OUTPUT noise at 0.05	CƐAL	TTT	(10,20)	100	924573.34
model3.dot INPUT noise at 0.1	CƐAL	TTT	(20,30)	31	37907809.55
model3.dot	CƐAL	TTT	(10,20)	100	2572752.19
OUTPUT noise at 0.1	MAT	TTT	(20,30)	76	1730604
BitVise.dot	CƐAL	TTT	(5,10)	100	11049320.81
INPUT noise at 0.01	MAT	$L^\star$	(10,20)	100	15036748.06
BitVise.dot	CƐAL	KV	(5,10)	100	11392077.23
OUTPUT noise at 0.01	MAT	TTT	(10,20)	100	14827071.84
BitVise.dot	CƐAL	KV	(20,30)	91	69346667.59
INPUT noise at 0.05	MAT	$L^\star$	(20,30)	1	6870614
BitVise.dot OUTPUT noise at 0.05	CƐAL	KV	(20,30)	77	77534701.29

Bibliography

[Mea55] George H. Mealy. “A Method for Synthesizing Sequential Circuits”. In: *The Bell System Technical Journal* 34.5 (Sept. 1955), pp. 1045–1079. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[Kle56] S. C. Kleene. “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies. (AM-34)*. Ed. by C. E. Shannon and J. McCarthy. Princeton University Press, Dec. 31, 1956, pp. 3–42. ISBN: 978-1-4008-8261-8. DOI: 10.1515/9781400882618-002.

[Ten+57] Stanley Tennenbaum, Anil Nerode, Josephine Mehlberg, and John Myhill. *Fundamental Concepts in the Theory of Systems*. ASTIA Document. Wright Air Development Center, Air Research and Development Command, United States Air Force, 1957.

[Ner58] A. Nerode. “Linear Automaton Transformations”. In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544. ISSN: 0002-9939, 1088-6826. DOI: 10.1090/S0002-9939-1958-0135681-9. URL: https://www.ams.org/proc/1958-009-04/S0002-9939-1958-0135681-9/ (visited on 01/06/2022).

[CS63] Noam Chomsky and M. Schützenberger. “The Algebraic Theory of Context-Free Languages”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 35. Elsevier, 1963, pp. 118–161. ISBN: 978-0-444-53400-2. DOI: 10.1016/S0049-237X(08)72023-8.

[Flo67] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science*. Ed. by J. T. Schwartz. Vol. 19. Proceed-

ings of Symposia in Applied Mathematics. Providence, RI: American Mathematical Society, 1967, pp. 19–32. ISBN: 0-8218-6728-8.

[Gol67] E Mark Gold. “Language Identification in the Limit”. In: *Information and Control* 10.5 (May 1, 1967), pp. 447–474. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)91165-5.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/363235.363259.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

[Sco70] Dana S. Scott. *Outline of a Mathematical Theory of Computation*. Technical Monograph PRG-2. Oxford, England: Oxford University Computing Laboratory, Programming Research Group, Nov. 1970. URL: https://www.cs.ox.ac.uk/files/3222/PRG02.pdf.

[BF72] A. W. Biermann and J. A. Feldman. “On the Synthesis of Finite-State Machines from Samples of Their Behavior”. In: *IEEE Transactions on Computers* C-21.6 (June 1972), pp. 592–597. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009015.

[Dij75] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360933.360975.

[Lav75] Mark S. Laventhal. “Verifying Programs Which Operate on Data Structures”. In: *ACM SIGPLAN Notices* 10.6 (June 1975), pp. 420–426. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/390016.808465.

[HR77] Harry B. Hunt and Daniel J. Rosenkrantz. “On Equivalence and Containment Problems for Formal Languages”. In: *Journal of the ACM*

24.3 (July 1977), pp. 387–396. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322017.322020.

[BJ78] Dines Bjørner and Cliff B. Jones, eds. *The Vienna Development Method: The Meta-Language*. Red. by G. Goos, J. Hartmanis, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, and N. Wirth. Vol. 61. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978. ISBN: 978-3-540-08766-3 978-3-540-35836-7. DOI: 10.1007/3-540-08766-4.

[Coo78] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM Journal on Computing* 7.1 (Feb. 1978), pp. 70–90. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0207005.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. New York, NY: Springer New York, 1979. ISBN: 978-0-387-90433-7 978-1-4612-6228-2. DOI: 10.1007/978-1-4612-6228-2.

[Lon+79] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. "Proof Rules for the Programming Language Euclid". In: *Program Construction*. Ed. by Friedrich L. Bauer, Manfred Broy, E. W. Dijkstra, S. L. Gerhart, D. Gries, M. Griffiths, J. V. Guttag, J. J. Horning, S. S. Owicki, C. Pair, H. Partsch, P. Pepper, M. Wirsing, and H. Wössner. Vol. 69. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 133–163. ISBN: 978-3-540-09251-3 978-3-540-35312-6. DOI: 10.1007/BFb0014666.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. "Specification Language". In: *On the Construction of Programs*. Ed. by A. M. Macnaghten and R. M. McKeag. Cambridge University Press, 1980, pp. 343–410. ISBN: 0-521-23090-X.

[Pos80] Jon Postel. *DoD Standard Transmission Control Protocol*. RFC 761. RFC Editor, Jan. 1980. DOI: 10.17487/RFC0761.

[Jon83] C. B. Jones. "Tentative Steps toward a Development Method for Interfering Programs". In: *ACM Transactions on Programming Languages and Systems* 5.4 (Oct. 1983), pp. 596–619. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/69575.69577.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Transactions on Programming Languages and Systems* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/5397.5399.

[Ang87] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation* 75.2 (Nov. 1987), pp. 87–106. ISSN: 08905401. DOI: 10.1016/0890-5401(87)90052-6.

[AL88] Dana Angluin and Philip Laird. "Learning From Noisy Examples". In: *Machine Learning* 2.4 (Apr. 1988), pp. 343–370. ISSN: 0885-6125, 1573-0565. DOI: 10.1023/A:1022873112823.

[Fuj+91] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering* 17.6 (June 1991), pp. 591–603. ISSN: 00985589. DOI: 10.1109/32.87284.

[Sak91] Yasubumi Sakakibara. "On Learning from Queries and Counterexamples in the Presence of Noise". In: *Information Processing Letters* 37.5 (Mar. 1991), pp. 279–284. ISSN: 00200190. DOI: 10.1016/0020-0190(91)90220-C.

[RS93] R.L. Rivest and R.E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Information and Computation* 103.2 (Apr. 1993), pp. 299–347. ISSN: 08905401. DOI: 10.1006/inco.1993.1021.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. “Case Study: Paris Metro Signaling System”. In: *IEEE Software* 11.1 (Jan. 1994), pp. 32–28. ISSN: 0740-7459. DOI: 10.1109/MS.1994.1279941.

[KV94] Michael J. Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Aug. 15, 1994. ISBN: 978-0-262-27686-3. DOI: 10.7551/mitpress/3897.001.0001.

[LY94] D. Lee and M. Yannakakis. “Testing Finite-State Machines: State Identification and Verification”. In: *IEEE Transactions on Computers* 43.3 (Mar. 1994), pp. 306–320. ISSN: 00189340. DOI: 10.1109/12.272431.

[Cro+95] Judith Crow, Ben Di Vito, Robyn Lutz, Larry Roberts, Martin Feather, and John Kelly. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner’s Companion*. Technical Guidebook NASA-GB-001-97/VOL2. Washington, DC: NASA Office of Safety and Mission Assurance, 1995. URL: https://ntrs.nasa.gov/citations/19980227975.

[BV96] Francesco Bergadano and Stefano Varricchio. “Learning Behaviors of Automata from Multiplicity and Equivalence Queries”. In: *SIAM Journal on Computing* 25.6 (Dec. 1996), pp. 1268–1280. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S009753979326091X.

[WJJ96] Jeannette Wing, Daniel Jackson, and Cliff B. Jones. “Formal Methods Light”. In: *Computer* 29.04 (Apr. 1996), pp. 20–22. ISSN: 1558-0814. DOI: 10.1109/MC.1996.10038.

[Eas+98] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. “Experiences Using Lightweight Formal Methods for Requirements Modeling”. In: *IEEE Transactions on Software Engineering* 24.1 (Jan. 1998), pp. 4–14. ISSN: 00985589. DOI: 10.1109/32.663994.

[McK98] William M. McKeeman. “Differential Testing for Software”. In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[PVY99] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. “Black Box Checking”. In: *Formal Methods for Protocol Engineering and Distributed Systems*. Ed. by Jianping Wu, Samuel T. Chanson, and Qiang Gao. Vol. 28. Boston, MA: Springer US, 1999, pp. 225–240. ISBN: 978-1-4757-5270-0 978-0-387-35578-8. DOI: 10.1007/978-0-387-35578-8_13.

[DLT01] François Denis, Aurélien Lemay, and Alain Terlutte. “Residual Finite State Automata”. In: *STACS 2001*. Ed. by Afonso Ferreira and Horst Reichel. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 144–157. ISBN: 978-3-540-41695-1 978-3-540-44693-4. DOI: 10.1007/3-540-44693-1_13.

[OS01] Arlindo L. Oliveira and João P.M. Silva. “Efficient Algorithms for the Inference of Minimum Size DFAs”. In: *Machine Learning* 44.1–2 (July 2001), pp. 93–119. ISSN: 0885-6125, 1573-0565. DOI: 10.1023/A:1010828029885.

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. “Adaptive Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 2280. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 357–370. ISBN: 978-3-540-43419-1 978-3-540-46002-2. DOI: 10.1007/3-540-46002-0_25.

[Jac02] Daniel Jackson. “Alloy: A Lightweight Object Modelling Notation”. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 2002), pp. 256–290. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/505145.505149.

[Bio03] Philippe Biondi. *Scapy: Packet Manipulation Library*. 2003. URL: https://scapy.net/ (visited on 01/22/2021).

[Mar+04] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. "Efficient Test-Based Model Generation for Legacy Reactive Systems". In: *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*. Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940). Sonoma Valley, CA, USA: IEEE, 2004, pp. 95–100. ISBN: 978-0-7803-8714-0. DOI: 10.1109/HLDVT.2004.1431246.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. "Finite State Machines for Strings over Infinite Alphabets". In: *ACM Transactions on Computational Logic* 5.3 (July 2004), pp. 403–435. ISSN: 1529-3785, 1557-945X. DOI: 10.1145/1013560.1013562.

[TV05] Deian Tabakov and Moshe Y. Vardi. "Experimental Evaluation of Classical Automata Constructions". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Geoff Sutcliffe and Andrei Voronkov. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 3835. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 396–411. ISBN: 978-3-540-30553-8 978-3-540-31650-3. DOI: 10.1007/11591191_28.

[Cha+08] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. "Verification of Evolving Software via Component Substitutability Analysis". In: *Formal Methods in System Design* 32.3 (June 2008), pp. 235–266. ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-008-0053-x.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 4963. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78799-0 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[Păs+08] Corina S. Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. "Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning". In: *Formal Methods in System Design* 32.3 (June 2008), pp. 175–205. ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-008-0049-6.

[Bol+09] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. "Angluin-Style Learning of NFA". In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1004–1009.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler, eds. *Handbook of Weighted Automata*. Monographs in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-01491-8 978-3-642-01492-5. DOI: 10.1007/978-3-642-01492-5.

[SG09] Muzammil Shahbaz and Roland Groz. "Inferring Mealy Machines". In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor,

Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 5850. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 207–222. ISBN: 978-3-642-05088-6 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_14.

[Sou+09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. "Formal Verification of Avionics Software Products". In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 5850. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 532–546. ISBN: 978-3-642-05088-6 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_34.

[ASV10] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. "Inference and Abstraction of the Biometric Passport". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 673–686. ISBN: 978-3-642-16558-0. DOI: 10.1007/978-3-642-16558-0_54.

[De 10] Colin De La Higuera. *Grammatical Inference: Learning Automata and Grammars*. 1st ed. Cambridge University Press, Apr. 1, 2010. ISBN: 978-0-521-76316-5 978-1-139-19465-5. DOI: 10.1017/CBO9781139194655.

[HV10] Marijn J. H. Heule and Sicco Verwer. "Exact DFA Identification Using SAT Solvers". In: *Grammatical Inference: Theoretical Results and Applications*. Ed. by José M. Sempere and Pedro García. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu

Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 6339. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 66–79. ISBN: 978-3-642-15487-4 978-3-642-15488-1. DOI: 10.1007/978-3-642-15488-1_7.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. "δ-Complete Decision Procedures for Satisfiability over the Reals". In: *Automated Reasoning*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 7364. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 286–300. ISBN: 978-3-642-31364-6 978-3-642-31365-3. DOI: 10.1007/978-3-642-31365-3_23.

[Goo12] Google. *Google QUIC*. 2012. URL: https://www.chromium.org/quic (visited on 01/25/2021).

[Jia+12] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. "Modeling and Verification of a Dual Chamber Implantable Pacemaker". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Cormac Flanagan and Barbara König. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 7214. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 188–203. ISBN: 978-3-642-28755-8 978-3-642-28756-5. DOI: 10.1007/978-3-642-28756-5_14.

[LN12] Martin Leucker and Daniel Neider. "Learning Minimal Deterministic Automata from Inexperienced Teachers". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mas-*

tering Change. Ed. by Tiziana Margaria and Bernhard Steffen. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 7609. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 524–538. ISBN: 978-3-642-34025-3 978-3-642-34026-0. DOI: 10.1007/978-3-642-34026-0_39.

[ARP13] F. Aarts, J. De Ruiter, and E. Poll. "Formal Models of Bank Cards for Free". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. Mar. 2013, pp. 461–468. DOI: 10.1109/ICSTW.2013.60.

[BMS13] Marcello M. Bonsangue, Stefan Milius, and Alexandra Silva. "Sound and Complete Axiomatizations of Coalgebraic Language Equivalence". In: *ACM Transactions on Computational Logic* 14.1 (Feb. 2013), pp. 1–52. ISSN: 1529-3785, 1557-945X. DOI: 10.1145/2422085.2422092.

[LU13] Tyll Leyden and Unknown (original, 2005). *TCP Three-Way Handshake*. 2013. URL: https://commons.wikimedia.org/wiki/File:Tcp-handshake.svg.

[Moy+13] Yannick Moy, Emmanuel Ledinot, Herve Delseny, Virginie Wiels, and Benjamin Monate. "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience". In: *IEEE Software* 30.3 (May 2013), pp. 50–57. ISSN: 0740-7459. DOI: 10.1109/MS.2013.43.

[Win+13] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. "Active Continuous Quality Control". In: *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*. Comparch '13: Federated Events on Component-Based Software Engineering and Software Ar-

chitecture. Vancouver British Columbia Canada: ACM, June 17, 2013, pp. 111–120. ISBN: 978-1-4503-2122-8. DOI: 10.1145/2465449.2465469.

[IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. Cham: Springer International Publishing, 2014, pp. 307–322. ISBN: 978-3-319-11163-6 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_26.

[vHee14] Gerco van Heerdt. "Efficient Inference of Mealy Machines". BA thesis. Radboud University Nijmegen, 2014.

[BM15] Borja Balle and Mehryar Mohri. "Learning Weighted Automata". In: *Algebraic Informatics*. Ed. by Andreas Maletti. Vol. 9270. Cham: Springer International Publishing, 2015, pp. 1–21. ISBN: 978-3-319-23020-7 978-3-319-23021-4. DOI: 10.1007/978-3-319-23021-4_1.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Vol. 9058. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 978-3-319-17523-2 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_1.

[HCM15] Michael D. Harrison, José Creissac Campos, and Paolo Masci. "Reusing Models and Properties in the Analysis of Similar Interactive Devices". In: *Innovations in Systems and Software Engineering* 11.2 (June 2015), pp. 95–111. ISSN: 1614-5046, 1614-5054. DOI: 10.1007/s11334-013-0201-3.

[Sme+15] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N. Jansen. "Applying Automata Learning to Embedded Control Software".

In: *Formal Methods and Software Engineering*. Ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Cham: Springer International Publishing, 2015, pp. 67–83. ISBN: 978-3-319-25422-7 978-3-319-25423-4. DOI: 10.1007/978-3-319-25423-4_5.

[Arg+16] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 1690–1701. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978383.

[Cas+16] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. "Active Learning for Extended Finite State Machines". In: *Formal Aspects of Computing* 28.2 (Apr. 1, 2016), pp. 233–263. ISSN: 0934-5043. DOI: 10.1007/s00165-016-0355-5.

[De 16] Joeri De Ruiter. "A Tale of the OpenSSL State Machine: A Large-Scale Black-Box Analysis". In: *Secure IT Systems*. Ed. by Billy Bob Brumley and Juha Röning. Vol. 10014. Cham: Springer International Publishing, 2016, pp. 169–184. ISBN: 978-3-319-47559-2 978-3-319-47560-8. DOI: 10.1007/978-3-319-47560-8_11.

[FJV16] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. "Combining Model Learning and Model Checking to Analyze TCP Implementations". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 454–471. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_25.

[Bha+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf

Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. "Everest: Towards a Verified, Drop-in Replacement of HTTPS". In: *LIPIcs, Volume 71, SNAPL 2017* 71 (2017). Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, 1:1–1:12. ISSN: 1868-8969. DOI: 10.4230/LIPICS.SNAPL.2017.1.

[DD17] Samuel Drews and Loris D'Antoni. "Learning Symbolic Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 173–189. ISBN: 978-3-662-54576-8 978-3-662-54577-5. DOI: 10.1007/978-3-662-54577-5_10.

[Fit+17] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. "Model Learning and Model Checking of SSH Implementations". In: *SPIN*. ACM, 2017, pp. 142–151. DOI: 10.1145/3092282.3092289.

[Moe+17] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. "Learning Nominal Automata". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Jan. 2017, pp. 613–625. DOI: 10.1145/3009837.3009879. arXiv: 1607.06268 [cs].

[Vaa17] Frits Vaandrager. "Model Learning". In: *Communications of the ACM* 60.2 (Jan. 23, 2017), pp. 86–95. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2967606.

[Wie17] Wojciech Wieczorek. *Grammatical Inference*. Vol. 673. Studies in Computational Intelligence. Cham: Springer International Publishing,

2017. ISBN: 978-3-319-46800-6 978-3-319-46801-3. DOI: 10.1007/978-3-319-46801-3.

[Cim+18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. "Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions". In: *ACM Transactions on Computational Logic* 19.3 (July 2018), pp. 1–52. ISSN: 1529-3785, 1557-945X. DOI: 10.1145/3230639. (Visited on 04/24/2026).

[Clo18] Cloudflare. *Cloudflare Quiche*. Cloudflare, Sept. 2018. URL: https://github.com/cloudflare/quiche (visited on 01/25/2021).

[Fac18] Facebook. *Facebook Mvfst*. Facebook Incubator, 2018. URL: https://github.com/facebookincubator/mvfst (visited on 01/25/2021).

[HMV18] David Huistra, Jeroen Meijer, and Jaco Van De Pol. "Adaptive Learning for Learn-Based Regression Testing". In: *Formal Methods for Industrial Critical Systems*. Ed. by Falk Howar and Jiří Barnat. Vol. 11119. Cham: Springer International Publishing, 2018, pp. 162–177. ISBN: 978-3-030-00243-5 978-3-030-00244-2. DOI: 10.1007/978-3-030-00244-2_11.

[OHe18] Peter W. O'Hearn. "Continuous Reasoning: Scaling the Impact of Formal Methods". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. Oxford United Kingdom: ACM, July 2018, pp. 13–25. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209109.

[PDB18] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. "Observing the Evolution of QUIC Implementations". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. CoNEXT '18: The 14th International Conference on Emerging Networking EXperiments and Technologies. Heraklion Greece: ACM,

Dec. 4, 2018, pp. 8–14. ISBN: 978-1-4503-6082-1. DOI: 10.1145/3284850.3284852.

[RSW18] Felix Rath, Daniel Schemmel, and Klaus Wehrle. "Interoperability-Guided Testing of QUIC Implementations Using Symbolic Execution". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. CoNEXT '18: The 14th International Conference on Emerging Networking EXperiments and Technologies. Heraklion Greece: ACM, Dec. 4, 2018, pp. 15–21. ISBN: 978-1-4503-6082-1. DOI: 10.1145/3284850.3284853.

[SFV18] Rick Smetsers, Paul Fiterău-Broştean, and Frits Vaandrager. "Model Learning as a Satisfiability Modulo Theories Problem". In: *Language and Automata Theory and Applications*. Ed. by Shmuel Tomi Klein, Carlos Martín-Vide, and Dana Shapira. Vol. 10792. Cham: Springer International Publishing, 2018, pp. 182–194. ISBN: 978-3-319-77312-4 978-3-319-77313-1. DOI: 10.1007/978-3-319-77313-1_14.

[Bis+19] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. "Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API". In: *Journal of the ACM* 66.1 (Feb. 28, 2019), pp. 1–77. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/3243650.

[DAn+19] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. "Symbolic Register Automata". In: *Computer Aided Verification*. Ed. by I. Dillig and S. Tasiran. Vol. 11561. Cham: Springer International Publishing, 2019, pp. 3–21. ISBN: 978-3-030-25539-8 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_1.

[DMD19] Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso Da Silva Simao. "Learning to Reuse: Adaptive Model Learning for Evolving Systems". In: *Integrated Formal Methods*. Ed. by Wolfgang

Ahrendt and Silvia Lizeth Tapia Tarifa. Vol. 11918. Cham: Springer International Publishing, 2019, pp. 138–156. ISBN: 978-3-030-34967-7 978-3-030-34968-4. DOI: 10.1007/978-3-030-34968-4_8.

[Dis+19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. "Scaling Static Analyses at Facebook". In: *Communications of the ACM* 62.8 (July 24, 2019), pp. 62–70. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3338112.

[MZ19] Kenneth L. McMillan and Lenore D. Zuck. "Formal Specification and Testing of QUIC". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19: ACM SIGCOMM 2019 Conference. Beijing China: ACM, Aug. 19, 2019, pp. 227–240. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342087.

[Nei+19] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. "Benchmarks for Automata Learning and Conformance Testing". In: *Models, Mindsets, Meta: The What, the How, and the Why Not?* Ed. by Tiziana Margaria, Susanne Graf, and Kim G. Larsen. Vol. 11200. Cham: Springer International Publishing, 2019, pp. 390–416. ISBN: 978-3-030-22347-2 978-3-030-22348-9. DOI: 10.1007/978-3-030-22348-9_23.

[Ter+19] Maurice H. Ter Beek, Arne Borälv, Alessandro Fantechi, Alessio Ferrari, Stefania Gnesi, Christer Löfving, and Franco Mazzanti. "Adopting Formal Methods in an Industrial Setting: The Railways Case". In: *Formal Methods – The Next 30 Years*. Ed. by Maurice H. Ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Cham: Springer International Publishing, 2019, pp. 762–772. ISBN: 978-3-030-30941-1 978-3-030-30942-8. DOI: 10.1007/978-3-030-30942-8_46.

[Fit+20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. "Analysis of {DTLS} Implementations Using Protocol State Fuzzing". In: 29th {USENIX} Secu-

rity Symposium ({USENIX} Security 20). 2020, pp. 2523–2540. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean (visited on 07/02/2021).

[GPP20] Vidhi Goel, Rui Paulo, and Christoph Paasch. "Testing QUIC with Packetdrill". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. SIGCOMM '20: Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. Virtual Event USA: ACM, Aug. 10, 2020, pp. 1–7. ISBN: 978-1-4503-8047-8. DOI: 10.1145/3405796.3405825.

[HJM20] Léo Henry, Thierry Jéron, and Nicolas Markey. "Active Learning of Timed Automata with Unobservable Resets". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Nathalie Bertrand and Nils Jansen. Vol. 12288. Cham: Springer International Publishing, 2020, pp. 144–160. ISBN: 978-3-030-57627-1 978-3-030-57628-8. DOI: 10.1007/978-3-030-57628-8_9.

[IT20] Jana Iyengar and Martin Thomson. *QUIC: A UDP-based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-29. Internet Engineering Task Force / Internet Engineering Task Force, June 10, 2020. 187 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/29/.

[Rei+20] Alastair Reid, Shaked Flur, Luke Church, Sarah de Haas, Maritza Johnson, and Ben Laurie. "Towards Making Formal Methods Normal: Meeting Developers Where They Are". In: *HATRA 2020: Human Aspects of Types and Reasoning Assistants*. 2020. URL: https://arxiv.org/abs/2010.16345.

[SB20] Michal Soucha and Kirill Bogdanov. "Observation Tree Approach: Active Learning Relying on Testing". In: *The Computer Journal* 63.9

(Aug. 20, 2020), pp. 1298–1310. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/bxz056.

[Tik20] Dmitri Tikhonov. *Let Server Abort on Post-Retry Packet Number Reset*. 2020. URL: https://github.com/quicwg/base-drafts/pull/3990 (visited on 01/22/2021).

[Van+20] Gerco Van Heerdt, Clemens Kupke, Jurriaan Rot, and Alexandra Silva. "Learning Weighted Automata over Principal Ideal Domains". In: *Foundations of Software Science and Computation Structures*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Cham: Springer International Publishing, 2020, pp. 602–621. ISBN: 978-3-030-45230-8 978-3-030-45231-5. DOI: 10.1007/978-3-030-45231-5_31.

[VSS20] Gerco Van Heerdt, Matteo Sammartino, and Alexandra Silva. "Learning Automata with Side-Effects". In: *Coalgebraic Methods in Computer Science*. Ed. by Daniela Petrişan and Jurriaan Rot. Vol. 12094. Cham: Springer International Publishing, 2020, pp. 68–89. ISBN: 978-3-030-57200-6 978-3-030-57201-3. DOI: 10.1007/978-3-030-57201-3_5.

[Vil+20] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. "CacheQuery: Learning Replacement Policies from Hardware Caches". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 519–532. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386008.

[ASZ21] Shahbaz Ali, Hailong Sun, and Yongwang Zhao. "Model Learning: A Survey of Foundations, Tools and Applications". In: *Frontiers of Computer Science* 15.5 (Oct. 2021), p. 155210. ISSN: 2095-2228, 2095-2236. DOI: 10.1007/s11704-019-9212-z.

[Bor+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. "Using

Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. Virtual Event Germany: ACM, Oct. 2021, pp. 836–850. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483540.

[Del+21] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. "A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2021, pp. 1162–1178. ISBN: 978-1-7281-8934-5. DOI: 10.1109/SP40001.2021.00039.

[Fer+21] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. "Prognosis: Closed-Box Analysis of Network Protocol Implementations". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21: ACM SIGCOMM 2021 Conference. Virtual Event USA: ACM, Aug. 9, 2021, pp. 762–774. ISBN: 978-1-4503-8383-7. DOI: 10.1145/3452296.3472938.

[IT21] Jana Iyengar and Martin Thomson. *QUIC: A UDP-based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, May 2021. DOI: 10.17487/RFC9000.

[Khm+21] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. "Property-Directed Verification and Robustness Certification of Recurrent Neural Networks". In: *Automated Technology for Verification and Analysis*. Ed. by Zhe Hou and Vijay Ganesh. Vol. 12971. Cham: Springer International Publishing, 2021, pp. 364–380. ISBN: 978-3-030-88884-8 978-3-030-88885-5. DOI: 10.1007/978-3-030-88885-5_24.

[Van+21] Gerco Van Heerdt, Tobias Kappé, Jurriaan Rot, and Alexandra Silva. "Learning Pomset Automata". In: *Foundations of Software Science and Computation Structures*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Cham: Springer International Publishing, 2021, pp. 510–530. ISBN: 978-3-030-71994-4 978-3-030-71995-1. DOI: 10.1007/978-3-030-71995-1_26.

[All22] Alloy Tools. *Alloy: Case Study Applications*. 2022. URL: https://alloytools.org/citations/case-studies.html (visited on 02/27/2026).

[ABK22] Shaull Almagor, Udi Boker, and Orna Kupferman. "What's Decidable about Weighted Automata?" In: *Information and Computation* 282 (Jan. 2022), p. 104651. ISSN: 08905401. DOI: 10.1016/j.ic.2020.104651.

[Bat+22] Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. "Weighted Programming: A Programming Paradigm for Specifying Mathematical Models". In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA1 Apr. 29, 2022), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3527310.

[FVS22] Tiago Ferreira, Gerco Van Heerdt, and Alexandra Silva. "Tree-Based Adaptive Model Learning". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra Van Den Bos. Vol. 13560. Cham: Springer Nature Switzerland, 2022, pp. 164–179. ISBN: 978-3-031-15628-1 978-3-031-15629-8. DOI: 10.1007/978-3-031-15629-8_10.

[Khm+22] Igor Khmelnitsky, Serge Haddad, Lina Ye, Benoît Barbot, Benedikt Bollig, Martin Leucker, Daniel Neider, and Rajarshi Roy. "Analyzing Robustness of Angluin's L* Algorithm in Presence of Noise". In: *Electronic Proceedings in Theoretical Computer Science* 370 (Sept. 20, 2022), pp. 81–96. ISSN: 2075-2180. DOI: 10.4204/EPTCS.370.6.

[Nas+22] Amir Naseredini, Martin Berger, Matteo Sammartino, and Shale Xiong. "ALARM: Active LeArning of Rowhammer Mitigations". In: *Proceedings of the 11th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '22: Hardware and Architectural Support for Security and Privacy. Chicago IL USA: ACM, Oct. 2022, pp. 1–9. ISBN: 978-1-4503-9871-8. DOI: 10.1145/3569562.3569563.

[TAL22] Martin Tappler, Bernhard K. Aichernig, and Florian Lorber. "Timed Automata Learning via SMT Solving". In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Cham: Springer International Publishing, 2022, pp. 489–507. ISBN: 978-3-031-06772-3 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_26.

[Vaa+22] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. "A New Approach for Active Automata Learning Based on Apartness". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Cham: Springer International Publishing, 2022, pp. 223–243. ISBN: 978-3-030-99523-2 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_12.

[Van+22] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. "Verifying Dynamic Trait Objects in Rust". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE '22: 44th International Conference on Software Engineering. Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 321–330. ISBN: 978-1-4503-9226-6. DOI: 10.1145/3510457.3513031.

[ABA23] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. "Formal Methods for Network Performance Analysis". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI*

23). Boston, MA: USENIX Association, Apr. 2023, pp. 645–661. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/tahmasbi.

[Fer+23] Tiago Ferreira, Léo Henry, Raquel Fernandes Da Silva, and Alexandra Silva. "Conflict-Aware Active Automata Learning". In: *Electronic Proceedings in Theoretical Computer Science* 390 (Sept. 30, 2023), pp. 150–167. ISSN: 2075-2180. DOI: 10.4204/EPTCS.390.10.

[Kum23] Rahul Kumar. *How Open Source Projects Are Using Kani to Write Better Software in Rust*. Nov. 6, 2023. URL: https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/ (visited on 02/27/2026).

[Moe+23] Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. "Automata Learning with an Incomplete Teacher". In: *LIPIcs, Volume 263, ECOOP 2023* 263 (2023). Ed. by Karim Ali and Guido Salvaneschi, 21:1–21:30. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECOOP.2023.21.

[VEB23] Frits Vaandrager, Masoud Ebrahimi, and Roderick Bloem. "Learning Mealy Machines with One Timer". In: *Information and Computation* 295 (Dec. 2023), p. 105013. ISSN: 08905401. DOI: 10.1016/j.ic.2023.105013.

[Wag23] Masaki Waga. "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization". In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 3–26. ISBN: 978-3-031-37706-8. DOI: 10.1007/978-3-031-37706-8_1.

[ATW24] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. "Benchmarking Combinations of Learning and Testing Algorithms for Automata

Learning". In: *Formal Aspects of Computing* 36.1 (Mar. 31, 2024), pp. 1–37. ISSN: 0934-5043, 1433-299X. DOI: 10.1145/3605360.

[Bun+24] Alex Buna-Marginean, Vincent Cheval, Mahsa Shirmohammadi, and James Worrell. "On Learning Polynomial Recursive Programs". In: *Proceedings of the ACM on Programming Languages* 8 (POPL Jan. 2, 2024), pp. 1001–1027. ISSN: 2475-1421. DOI: 10.1145/3632876.

[Die+24] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. "Scalable Tree-based Register Automata Learning". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14571. Cham: Springer Nature Switzerland, 2024, pp. 87–108. ISBN: 978-3-031-57248-7 978-3-031-57249-4. DOI: 10.1007/978-3-031-57249-4_5.

[Jia+24] Shihao Jiang, Yu Zhang, Junqiang Li, Hongfang Yu, Long Luo, and Gang Sun. *A Survey of Network Protocol Fuzzing: Model, Techniques and Directions*. Feb. 27, 2024. DOI: 10.48550/arXiv.2402.17394. arXiv: 2402.17394 [cs]. Pre-published.

[KJR24] Loes Kruger, Sebastian Junges, and Jurriaan Rot. "Small Test Suites for Active Automata Learning". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14571. Cham: Springer Nature Switzerland, 2024, pp. 109–129. ISBN: 978-3-031-57248-7 978-3-031-57249-4. DOI: 10.1007/978-3-031-57249-4_6.

[Pas+24] Clemente Pasti, Talu Karagöz, Franz Nowak, Anej Svete, Reda Boumasmoud, and Ryan Cotterell. "An L* Algorithm for Deterministic Weighted Regular Languages". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. Miami, Florida, USA: Association for Computational Lin-

guistics, 2024, pp. 8197–8210. DOI: 10.18653/v1/2024.emnlp-main.468.

[Ari+25] Quentin Aristote, Sam Van Gool, Daniela Petrişan, and Mahsa Shirmohammadi. "Learning Weighted Automata over Number Rings, Concretely and Categorically". In: *2025 40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2025 40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). Singapore: IEEE, June 23, 2025, pp. 417–430. ISBN: 979-8-3315-7900-5. DOI: 10.1109/LICS65433.2025.00038.

[Bar+25] Celeste Barnaby, Qiaochu Chen, Ramya Ramalingam, Osbert Bastani, and Işıl Dillig. "Active Learning for Neurosymbolic Program Synthesis". In: *Proceedings of the ACM on Programming Languages* 9 (OOPSLA2 Oct. 9, 2025), pp. 1455–1483. ISSN: 2475-1421. DOI: 10.1145/3763102.

[DJ25] Laure Daviaud and Marianne Johnson. "Feasability of Learning Weighted Automata on a Semiring". In: *Logical Methods in Computer Science* Volume 21, Issue 3 (Aug. 7, 2025), p. 13612. ISSN: 1860-5974. DOI: 10.46298/lmcs-21(3:15)2025.

[FRG25] Eve Fernando, Sasha Rubin, and Rahul Gopinath. *Example-Free Learning of Regular Languages with Prefix Queries*. Apr. 2, 2025. DOI: 10.48550/arXiv.2504.02170. arXiv: 2504.02170 [cs]. Pre-published.

[FO25] Michał Fica and Jan Otop. *Active Automata Learning with Advice*. Aug. 14, 2025. DOI: 10.48550/arXiv.2508.10535. arXiv: 2508.10535 [cs]. Pre-published.

[FHS25] Markus Frohme, Falk Howar, and Bernhard Steffen. "LearnLib: 10 Years Later". In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Vol. 15934. Cham: Springer Nature Switzerland,

2025, pp. 141–160. ISBN: 978-3-031-98684-0 978-3-031-98685-7. DOI: 10.1007/978-3-031-98685-7_7.

[HBC25] Eric Hsiung, Joydeep Biswas, and Swarat Chaudhuri. "Automata Learning from Preference and Equivalence Queries". In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Vol. 15934. Cham: Springer Nature Switzerland, 2025, pp. 104–126. ISBN: 978-3-031-98684-0 978-3-031-98685-7. DOI: 10.1007/978-3-031-98685-7_5.

[KJR25] Loes Kruger, Sebastian Junges, and Jurriaan Rot. "State Matching and Multiple References in Adaptive Active Automata Learning". In: *Formal Methods*. Ed. by André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi. Vol. 14933. Cham: Springer Nature Switzerland, 2025, pp. 267–284. ISBN: 978-3-031-71161-9 978-3-031-71162-6. DOI: 10.1007/978-3-031-71162-6_14.

[Moe+25] Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. "Active Learning of Symbolic NetKAT Automata". In: *Proceedings of the ACM on Programming Languages* 9 (PLDI June 10, 2025), pp. 1119–1142. ISSN: 2475-1421. DOI: 10.1145/3729295.

[Rot+25] Noga H. Rotman, Tiago Ferreira, Hila Peleg, Mark Silberstein, and Alexandra Silva. *RFSeek and Ye Shall Find*. Under review at ANRW. Sept. 12, 2025. DOI: 10.48550/arXiv.2509.10216. arXiv: 2509.10216 [cs.NI]. Pre-published.

[Ter+25] Maurice H. Ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo Ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. "Formal Methods in Industry". In: *Formal Aspects of Computing* 37.1 (Mar. 31, 2025), pp. 1–38. ISSN: 0934-5043, 1433-299X. DOI: 10.1145/3689374.

[Bru+26] Véronique Bruyère, Bharat Garhewal, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. "Active Learning of Mealy Machines with Timers". In: *Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems*. Ed. by Pavithra Prabhakar and Andrea Vandin. Vol. 16143. Cham: Springer Nature Switzerland, 2026, pp. 42–61. ISBN: 978-3-032-05791-4 978-3-032-05792-1. DOI: 10.1007/978-3-032-05792-1_3.

[Fer26] Tiago Ferreira. *Automata Learning for Lightweight Formal Methods*. Zenodo, 2026. DOI: 10.5281/ZENODO.21806344.

[FBS26] Tiago Ferreira, Kevin Batz, and Alexandra Silva. "SMT-Based Active Learning of Weighted Automata". In: *Computer Aided Verification*. Ed. by Eva Darulova, Anthony W. Lin, and Philipp Rümmer. Vol. 16683. Cham: Springer Nature Switzerland, 2026, pp. 307–330. DOI: 10.1007/978-3-032-32526-6_15.

[For+26] Sophie Fortz, Fatemeh Ghassemi, Léo Henry, Falk Howar, Thomas Neele, Jurriaan Rot, and Marnix Suilen. "A Research Agenda for Active Automata Learning". In: *International Journal on Software Tools for Technology Transfer* (Feb. 19, 2026). ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-026-00839-z.

[Kru+26] Loes Kruger, Paul Kobialka, Andrea Pferscher, Einar Broch Johnsen, Sebastian Junges, and Jurriaan Rot. *Incremental Fingerprinting in an Open World*. Jan. 29, 2026. DOI: 10.48550/arXiv.2601.21680. arXiv: 2601.21680 [cs]. Pre-published.

[Suá+26] Emmanuel Suárez Acevedo, Tiago Ferreira, Kevin Batz, Oliver Bøving, Nate Foster, and Alexandra Silva. "Weighted NetKAT: A Programming Language for Quantitative Network Verification". In: *Proceedings of the ACM on Programming Languages* 10 (PLDI June 8, 2026), pp. 1788–1811. ISSN: 2475-1421. DOI: 10.1145/3808318.