Dipl.-Ing. Martin Tappler, BSc.

Learning-Based Testing in Networked Environments in the Presence of Timed and Stochastic Behaviour

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Main Supervisor
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institute of Software Technology (IST)

Co-Supervisor
Univ.-Prof. Roderick Bloem, Ph.D.
Institute of Applied Information Processing and Communications (IAIK)

External Reviewer and Examiner
Prof. Kim Guldstrand Larsen, Ph.D.
Department of Computer Science, Aalborg University

Graz, November 21, 2019

Dipl.-Ing. Martin Tappler, BSc.

Lernbasiertes Testen in vernetzten Umgebungen unter dem Einfluss zeitgesteuerten und stochastischen Verhaltens

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Hauptbetreuer
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institut für Softwaretechnologie (IST)

Co-Betreuer
Univ.-Prof. Roderick Bloem, Ph.D.
Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK)

Externer Gutachter und Prüfer
Prof. Kim Guldstrand Larsen, Ph.D.
Department of Computer Science, Aalborg University

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

Electronic software-based devices are ubiquitous and affect our everyday lives in various ways. They are used in entertainment as well as in safety-critical areas, such as driving-assistance systems. In addition to the growing importance of software, the last decades have seen a transition from large mainframe-style computers towards lightweight devices that solve complex tasks in collaboration. In recent years, the growing number of small-scale networked devices used in ordinary *things*, such as household appliances, has been dubbed the *internet of things (IoT)*. Hence, software systems serve important purposes on the one hand, while they form complex heterogeneous networks on the other hand. This calls for rigorous verification techniques addressing the specifics of today's networked software systems.

In this thesis, we propose the application of automata *learning-based testing* in this context. *Testing* is a well-established technique for error detection and *learning-based testing* is well-suited for our targeted application domain, due to its applicability in a so-called black-box setting. In this setting, we assume no knowledge about the internals of systems which is a realistic assumption in the IoT. Generally, learning-based testing incrementally extends its knowledge about the system under test through testing. It records the gained knowledge in models learned from test observations, which are used to derive further test cases.

The work performed towards this thesis spans two phases. In the exploratory phase, we combined and applied existing techniques in a case study on learning-based testing of implementations of the IoT protocol MQTT. This case study uncovered several violations of the MQTT specification, demonstrating that learning-based testing can be effective in networked environments. Additionally, shortcomings of the state of the art were discovered as well. We used these findings on shortcomings as the basis for the second phase, in which we contributed to the state of the art in various ways.

As a first step towards enhancing the applicability of learning-based testing in networked environments, we developed an approach for efficient conformance testing in active automata learning. Subsequent work focused on learning techniques for various types of systems, including stochastic, real-time, and hybrid systems, to broaden the applicability of learning-based testing. Our contributions in this context comprise: (1) a learning-based technique for testing stochastic systems with respect to reachability objectives, (2) an active test-based algorithm for learning stochastic system models, (3) a metaheuristic technique for learning real-time system models from test observations, and (4) a test-based machine learning process for hybrid system models. We implemented and evaluated all proposed learning and testing techniques. With that, we demonstrated that we are able to automatically generate near-optimal testing strategies in the presence of stochastic behaviour and that accurate models of stochastic systems can be learned via testing. Furthermore, our work demonstrates that metaheuristic techniques enable learning of real-time system models and that model-based testing combined with automata learning is able to generate good datasets for machine learning in the context of hybrid systems.

Keywords: Model-Based Testing, Automata Learning, Active Automata Learning, Learning-Based Testing, Model Inference, Model Learning.

Kurzfassung

Software-gesteuerte Geräte sind allgegenwärtig und beeinflussen unseren Alltag auf viele verschiedene Arten. Software wird sowohl in Unterhaltungselektronik als auch in sicherheitskritischen Bereichen, wie zum Beispiel Fahrassistenzsystemen, angewandt. In den letzten Jahren werden Software-basierte Systeme nicht nur wichtiger, sondern auch ihre Art und Struktur wandeln sich. Früher wurden Berechnungen von grossen Zentralrechnern ausgeführt, während heute kleine Geräte kollaborativ komplexe Aufgaben ausführen. Die zurzeit stark wachsende Zahl kleiner, vernetzter Recheneinheiten in alltäglichen *Dingen* wird als *Internet der Dinge (englisch: IoT)* bezeichnet. Die Zunahme der Wichtigkeit von Software erfordert Korrektheit, welche in komplexen, heterogene Netzwerken wie dem IoT aber schwierig zu erreichen ist. Diese Umstände verlangen nach rigorosen Verifikationstechniken, die auf die Spezifika der heutigen vernetzten Software-Systeme abgestimmt sind.

In dieser Dissertation wird *lernbasiertes Testen* präsentiert und angewandt. *Software-Testen* ist eine etablierte Technik für das Auffinden von Fehlern, während *lernbasiertes Testen* gut für vernetzte Systeme geeignet ist, da es in einem Black-Box-Setting anwendbar ist. In einem solchen Setting wird angenommen, dass kein Wissen über die Interna des zu testenden Systems verfügbar ist, was eine realistische Annahme im IoT darstellt. Die Funktionsweise von lernbasiertem Testen beruht darauf, dass inkrementell Wissen über das zu testende System gesammelt wird. Das gesammelte Wissen wird von gelernten Modellen repräsentiert, welche wiederum dazu dienen, neue Testfälle zu generieren.

Die hier präsentierte Arbeit umfasst zwei Phasen. In der ersten, explorativen Phase werden existierende Lern- und Testtechniken kombiniert und in einer Fallstudie zu lernbasiertem Testen angewandt. In dieser Fallstudie werden Implementierungen des IoT-Protokolls MQTT getestet und dadurch mehrere Spezifikationsverletzungen festgestellt. Damit wird zum einen demonstriert, dass lernbasiertes Testen effektiv Fehler in vernetzten Umgebungen auffinden kann und zum anderen werden Schwächen aktueller Lerntechniken aufgezeigt. Die gefundenen Schwächen dienen als Basis für die zweite Phase, in welcher gezielt Techniken erforscht werden, um diese Schwächen zu beseitigen und zum aktuellen Stand der Technik beizutragen.

Zuerst wird ein Ansatz zum effizienten Konformanztesten im aktivem Automaten-Lernen entwickelt, um die Anwendbarkeit von lernbasiertem Testen in vernetzten Umgebungen zu verbessern. Der Fokus nachfolgender Arbeiten richtet sich auf die Ausweitung des Anwendungsgebiets von testbasiertem Automaten-Lernen. Im Zuge dessen werden Techniken für verschiedene Arten von Systemen erforscht, wobei stochastische, Echtzeit- und hybride Systeme betrachtet werden. Die Forschungsbeiträge umfassen hierzu (1) eine lernbasierte Technik, um stochastische Systeme bezüglich der Erreichbarkeit spezifizierter Zustände zu testen, (2) einen aktiven testbasierten Algorithmus für das Lernen stochastischer Modelle, (3) eine metaheuristische Technik, um Modelle von Echtzeitsystemen zu lernen und (4) einen testbasierten Prozess für das maschinelle Lernen von Modellen, welche hybrides Systemverhalten repräsentieren. Alle präsentierten Lern- und Testtechniken wurden implementiert und experimentell evaluiert. Die Evaluierungsergebnisse zeigen, dass es durch lernbasiertes Testen möglich ist, automatisch effektive Teststrategien für stochastische Systeme zu generieren und, dass es basierend auf Testen möglich ist, akkurate Modelle stochastischer Systeme zu lernen. Des Weiteren demonstrieren durchgeführte Experimente, dass Modelle von Echtzeitsystemen durch metaheuristische Verfahren gelernt werden können. Durch die Kombination von modellbasiertem Testen und Automaten-Lernen ist es gelungen, Datensätze für präzises maschinelles Lernen von hybridem Systemverhalten zu generieren.

Schlagworte: Modellbasiertes Testen, Automaten-Lernen, Aktives Automaten-Lernen, Lernbasiertes Testen, Modell-Inferenz, Modell-Lernen.

Affidavit

Eidesstattliche Erklärung

Acknowledgements

I am grateful to my supervisor Bernhard Aichernig. His lectures introduced me to various fascinating areas including declarative programming, formal methods, and model-based testing. Moreover, Bernhard Aichernig introduced me to active automata learning, which offered many exciting challenges that I tackled in the course of the work on this thesis. He was always open to discuss new ideas and also to talk about sources of frustration that probably every PhD student faces after studying the literature and realising that *everything* has been done before. I also want to thank Kim Larsen for reviewing my thesis and for the opportunity to visit and work with him and his team in Aalborg.

I would like to thank the consortium of the Lead project "Dependable Internet of Things in Adverse Environments" headed by Kay Römer and the Graz University of Technology for funding this project that made the research presented within this thesis possible. The "Dependable Things"-project provided an excellent environment for performing my research. I would also like to thank my colleagues in this project, especially from the subproject "Dependable Composition" in which I participated. I had many fruitful and inspiring discussions with my co-supervisor Roderick Bloem, Masoud Ebrahimi, Tobias Schrank, Richard Schumi, Wolfgang Roth, and Franz Pernkopf. I want to address special thanks to Florian Lorber for providing helpful advice and being a great colleague and a nice guide during my stay in Aalborg.

I am also thankful to my friends and colleagues who helped me preparing this thesis by providing feedback on papers and on parts of this thesis. I want to thank Christian Burghard, Irena Ruprecht, Nina Ruderes, Andrea Pferscher and all my coauthors. Furthermore, I want to thank Andrea Pferscher and Felix Wallner, whose thesis projects I co-supervised, for their valuable contributions.

I would like to thank my partner Nina for her support, for helping me to rehearse my talks, for bearing with me in times of deadline pressure, and for celebrating successfully accepted papers with me. Finally, I want to thank my family, especially my aunt Elfi, my uncle Toni and most importantly my parents Gerhard and Helga, for providing me with a loving home and supporting me throughout my education.

Martin Tappler
Graz, Austria, November 21, 2019

Danksagung

Ich möchte meinem Betreuer Bernhard Aichernig danken. Seine Vorlesungen machten mich auf verschiedenste faszinierende Gebiete, wie die deklarative Programmierung, formale Methoden und modellbasiertes Testen, aufmerksam. Über dies hinaus hat Bernhard Aichernig mir aktives Automaten-Lernen näher gebracht, welches viele interessante Herausforderungen bot, die ich während der Arbeit an dieser Dissertation in Angriff genommen habe. Er war immer offen für Diskussionen über neue Ideen und auch über Quellen der Frustration, denen vermutlich alle Doktoratsstudierenden begegnen, genauer nach der Erkenntnis, dass bereits quasi *alles* erforscht wurde. Ich möchte auch Kim Larsen für das Begutachten der vorliegenden Dissertation danken und dafür, dass ich ihn und sein Team in Aalborg zur gemeinsamen Arbeit besuchen konnte.

Ich möchte mich beim Konsortium des Lead-Projekts "Dependable Internet of Things in Adverse Environments", unter der Leitung von Kay Römer, und bei der technischen Universität Graz für die Unterstützung und Finanzierung dieses Projekts bedanken, welches meine Forschung für die vorliegende Dissertation möglich gemacht hat. Das "Dependable Things"-Projekt hat mir ein ausgezeichnetes Umfeld für die Durchführung meiner Forschung geboten. Ich möchte auch meinen Kollegen in diesem Projekt, allen voran den Kollegen im Sub-Projekt "Dependable Composition", an dem ich teilgenommen habe, danken. Ich führte viele aufschlussreiche und inspirierende Diskussionen mit meinem Co-Betreuer Roderick Bloem, mit Masoud Ebrahimi, Tobias Schrank, Richard Schumi, Wolfgang Roth und Franz Pernkopf. Spezieller Dank gilt Florian Lorber, der mir hilfreiche Ratschläge gegeben hat und ein großartiger Kollege war und ist. Darüber hinaus wäre mein Besuch in Aalborg ohne ihn und seine Freunde nur halb so lustig gewesen.

Ich bin meinen Freunden und Kollegen dankbar, die mir während der Arbeit an dieser Dissertation geholfen haben, beispielsweise durch Feedback auf Konferenzbeiträge oder durch das Korrekturlesen von Teilen der Dissertation. In diesem Zusammenhang möchte ich Christian Burghard, Irena Ruprecht, Nina Ruderes, Andrea Pferscher und allen meinen Koautoren danken. Des weiteren möchte ich mich bei Andrea Pferscher und Felix Wallner, deren Abschlussprojekte ich mitbetreut habe, für ihre wertvollen Beiträge bedanken.

Ich möchte mich bei meiner Partnerin Nina für ihre Unterstützung bedanken – dafür, dass sie geholfen hat, meine Vorträge zu proben, dass sie für mich in stressreichen Zeiten vor Deadlines da ist and dafür, meine akzeptierten Konferenzbeiträge und Artikel mit mir zu feiern. Schlussendlich möchte ich mich bei meiner Familie, bei meiner Tante Elfi, meinem Onkel Toni und vor allem bei meinen Eltern Gerhard und Helga, dafür bedanken, dass sie mir ein liebevolles Zuhause geboten und mich während meiner Ausbildung unterstützt haben.

Martin Tappler
Graz, Österreich, 21. November 2019

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ioco	input-output conformance.
CAS	car alarm system.
CDF	cumulative distribution function.
CPS	cyber-physical system.
DFA	deterministic finite automaton.
DTMC	discrete time Markov chain.
EFSM	extended finite-state machine.
IoT	Internet of Things.
ISO	International Organization for Standardization.
LTL	linear temporal logic.
LTS	labelled transition system.
MAT	minimally adequate teacher.
MBT	model-based testing.
MDP	Markov decision process.
MQTT	Message Queuing Telemetry Transport.
NFA	nondeterministic finite automaton.
PAC	probably approximately correct.
PC	particle counter.
PCTL	probabilistic computation tree logic.
QoS	quality of service.
RNN	recurrent neural network.
SMC	statistical model-checking.
SMT	satisfiability modulo theories.
SUL	system under learning.

SUT system under test.

TA timed automaton.

TCP transmission control protocol.

TLS transport layer security.

TTS timed transition system.

UDP user datagram protocol.

1
Introduction

1.1 Motivation

Electronic software-based devices are ubiquitous and their number is ever-growing. They affect our daily lives in various ways, ranging from providing entertainment over enabling communication to performing safety-critical tasks, like controlling airbags in cars. This calls for a rigorous verification of software systems. In itself, the quest to verify that software is correct is not new. An early mathematical tool to verify programs, Hoare logic, was proposed by Antony Hoare fifty years ago in 1969 [140]. A year earlier, the "software crisis" was first considered an issue which refers to the increasing complexity of software that leads to errors in program code [238]. On the same occasion the term "software engineering" was coined as well, with the goal of establishing software engineering as an engineering discipline. Likewise, safety-critical software that may endanger lives is not new. In the eighties, for instance, the Therac-25 accidents caused deaths and serious injuries through overdoses in radiation therapy [189].

What has changed in recent years, though, is the increased number of software-based systems. Moreover, the nature of software-based systems shifted more towards networked systems that perform tasks in collaboration. Individual systems may be small and simple in these scenarios, such as temperature sensors, but complexity arises from composition in networked systems. Systems also tend to not only communicate with each other, but they interact with their environments as well. An adaptive cruise controller is an example of a safety-critical device that interacts with its environment. It controls the car and it also receives information from other software systems, such as the distance to other cars. Further examples are smart appliances that are connected to the internet, for instance, to provide remote-control features. Recently, the term **Internet of Things (IoT)** has been adopted to refer to electronic devices (things) that form networks to collaboratively perform tasks in our everyday lives. In conclusion, complexity is still increasing since the "software crisis" in the sixties and composition adds to that, while the correctness of software is becoming more and more important due to the ubiquitous nature of software.

In the short history of the IoT, we have already seen large-scale issues related to faulty or untrustworthy software systems. For example, in 2016 the Mirai botnet took over large numbers of IoT devices, such as webcams, to carry out distributed denial-of-service attacks [164]. In that case, the faults enabling these attacks were weak network configurations. Kolias et al. [164] state that IoT devices are easy targets for hackers "because they're constantly connected to the Internet and seemingly permeated with flaws". Another security-related issue concerning network communication is the SKIP-TLS vulnerability in the transport layer security (TLS) protocol [57]. This attack is enabled by the incorrect composition of state machines. A safety-related flaw of a feature of a Tesla® car was recently discovered only after deployment by a user [224]. The flaw caused the air conditioning to be turned off in "dog mode". This mode is actually supposed to regulate air condition to allow for dogs to be left alone in the car. Hence, the flaw endangered the life of these dogs. These examples demonstrate that it is necessary to verify that all parts of software including their interconnections are correctly implemented in order to ensure secure and safe operation. This even includes seemingly innocuous consumer electronics, like webcams.

Different from the software-engineering landscape from about fifty years ago, software testing is now well-accepted. While early research, such as Hoare logic [140], focused on proving programs correct, software testing nowadays is widely applied in practice [228]. It is accepted as a cost-effective means to detect errors in software. However, many testing approaches require either program code or an abstract system model as a basis for testing. As a consequence, they may not be applicable in the IoT, since commonly neither program code nor models are available for the countless devices in the IoT. Alternative approaches that do not rely on knowledge about system behaviour may be ineffective, as they are often unable to systematically cover the unknown behaviour. To address that, we propose the application of **learning-based testing** in networked environments, particularly in the IoT. This form of testing is applicable in a black-box setting, that is, in a setting without knowledge about the internals of the tested system and it incrementally learns models of the system's behaviour. These models help to systematically derive test cases for further testing and they can be analysed manually and through verification techniques, such as model checking.

1.2 Software Testing

Most text books on software testing contain definitions such as "The goal of a software testing is to find bugs." [228]. Additional objectives, like early bug detection, may also be given and what constitutes a bug may be further specified. For example, the IEEE standard glossary of Software Engineering Terminology distinguishes mistakes, faults, errors, and failures [150]. Mistakes refer to human actions, faults refer to incorrect parts of software, errors denote the difference between a correct value and a computed value, and failures are observable, incorrect results.[1]

In this thesis, we use more liberal definitions of bugs and software testing. The reason for this is that we perform learning-based testing and test-based learning, which will be explained below. Learning-based testing is a form of dynamic black-box testing [228]. In dynamic black-box testing, a tester provides inputs to the software, observes outputs, and checks whether those outputs are correct. The tester has no knowledge about the internals of the software, thus it is only possible to observe failures. Since a failure is generally caused by a fault, we will use the terms fault detection and failure detection mostly interchangeably. Moreover, we may also use the term bug to refer to failures.

In short, test-based learning automatically generates behavioural models from test observations. Its goal is to create faithful models regardless of the presence of bugs. Hence, it can be considered a form of exploratory testing [228], but it actually does not try to detect bugs. The goal of the testing applied in this thesis is generally to gain knowledge about the system at hand by providing inputs and observing outputs. This should not be confused with testing processes at "Level 0" described by Beizer [36, 50], referring to poorly implemented testing processes. Test-based learning is rather a component of a verification process that analyses learned models systematically, for instance, via model checking.

1.3 Model-Based Testing

In model-based testing, a behaviour model of the system under test (SUT) [273, 274, 296] serves as the basis for testing. Such a model explicitly describes the expected behaviour of the SUT and allows for the automatic generation of test cases. In addition to that, models serve as the basis for the decision

[1]Actually, the IEEE standard glossary contains another different definition of failure as well.

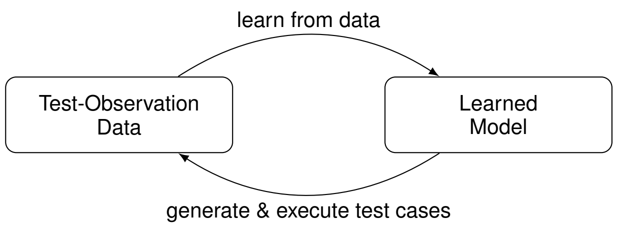


Figure 1.1: The interaction between model learning and model-based testing

on a verdict that is assigned to the execution of a test case. Models are also referred to as test oracles in this context. Therefore, models usually encode some requirements to be tested. The approach to assign verdicts varies between different testing techniques, but often a conformance relation captures the relation between the model and correct implementations [271].

As mentioned in Section 1.2, we generally perform a form of exploratory testing. This is true from a high-level point of view. At intermediate stages of learning, however, we aim at detecting new behaviour to extend a learned hypothesis model. With that, we essentially try to falsify the hypothesis model via model-based testing. Contrary to traditional model-based testing, we assume the SUT to be correct and test if the model is correct. More concretely, we test if the model conforms to the SUT with respect to some conformance relation by applying model-based testing techniques. Hence, our goal is to find faults in learned models. For more information on model-based testing, we refer to Section 2.4.

1.4 Learning-Based Testing and Test-Based Learning

In our survey titled "Model Learning and Model-Based Testing" [26], we discussed works in the intersection between model learning and model-based testing. We distinguished two types of research in this area: (1) *test-based learning* referring to (active) learning techniques that actively query systems via testing to gain knowledge about their behaviour and (2) *learning-based testing* which refers to works that apply model learning to generate models for model-based testing, or model-based verification in general. In this thesis, we apply similar definitions. Likewise, we concentrate on behavioural models of stateful systems, mainly in the form of automata.

To further clarify, with *test-based learning* we refer to techniques that learn models from observed test data. Such techniques are generally active techniques that actively query a system under learning through testing, but passive techniques that learn models from logged test observations are also possible.

Here, we use the term *learning-based testing* in a more general sense than in our survey [26]. As mentioned above, our survey classifies works to perform learning-based testing if they apply model learning to create models for model-based verification. Model-based verification may, for instance, involve model-based testing directly or model checking, which can be considered as exhaustive testing on model level. In this thesis, we extend this definition. We consider testing performed in the course of active test-based learning also as *learning-based testing*, because this form of testing is based on *learned* intermediate models. Hence, whenever we perform active test-based learning, we also perform learning-based testing. Figure 1.1 shows schematically the relation between model learning and model-based testing in an active setting. The former learns models from test data, whereas the latter derives test cases from learned models.

Thus, these two activities are often interconnected. We use these terms generally to emphasise either the learning aspect or the testing aspect. Hence, we use the term test-based learning to emphasise the learning aspect. In such cases, the goal is usually to create a behaviour model of a system under learning (SUL). In learning-based testing approaches, we emphasise the testing aspect which has varying goals. Potential goals are, for example, detecting specification violations, reliably provoking stochastic failures, or simply crashing the system under test (SUT). The acronyms system under learning (SUL) and SUT serve a similar purpose to put emphasis on either learning or testing.

1.5 Automata Learning

Now that we have discussed the form of testing we perform and introduced what test-based learning is, we want to discuss different forms of automata learning and their application. This discussion is based on our previous work on learning-based testing [18]. Note that while automata learning can learn from different sources of information, we generally consider test data as the main source of information.

Automata learning establishes the basis for model-based verification in a black-box setting by automatically learning automata models of black-box systems from observed test data. Data used for automata learning is usually given in the form of system traces, that is, sequences of system events. We commonly partition these events into input and output events. Unless otherwise noted, whenever we use the term learning, we refer to automata learning. Various other terms are used in the literature, such as model learning that we used in our survey on the topic [26], automata inference [95], regular inference [54], and regular extrapolation [134]. These terms basically refer to the same concepts, but emphasise different aspects. In this thesis, we mainly use the terms model learning and automata learning.

There are two main forms of model learning: *passive* learning and *active* learning. Passive learning learns from preexisting data such as system logs, while active learning actively queries the system that is examined to gain relevant information. In this thesis, we generally query systems via testing, but active learning may be applied in non-test-based settings as well. Non-test-based queries may, for instance, be implemented through model checking [87].

Noteworthy early examples of passive learning techniques are RPNI for regular languages [95, 225] and ALERGIA [74] for stochastic regular languages, which learn deterministic finite automata (DFA) and their stochastic counterparts, respectively. Both, RPNI and ALERGIA, apply a principle called state merging. These algorithms have been developed for grammatical inference and take text as input. In order to learn system models, system traces are usually considered to form (regular) languages. The k-tail algorithm [60] is an early example of a technique that directly addresses passive learning (synthesis) of finite-state machines from sample traces. More recent work based on the principle of state merging extends passive model learning to timed systems [279, 280, 282], Moore machines [119], and to stochastic systems involving non-deterministic choices [198, 199].

In contrast to passive learning, active learning approaches rely on the possibility to query relevant information. Angluin formalised this by introducing the minimally adequate teacher framework in her seminal work on the L^* algorithm [37]. This framework assumes the existence of a teacher that is able to answer two types of queries: membership queries and equivalence queries. Like RPNI, the original L^* algorithm learns DFA accepting some regular language. Hence, to learn a model of a software system via L^* it is necessary to assume that the system traces form a regular language. More concretely, membership queries in this framework basically check whether a given trace can be observed and equivalence queries check whether a hypothesised system model is equivalent to the system under learning. In practice, both queries are usually implemented via testing. Since the introduction of L^*, it has been adapted and extended to various types of systems, such as Mealy machines [200, 246], timed systems [126], and non-deterministic systems [161, 283].

1.6 Scope and Research Goals

1.6.1 Scope

Our focus lies on *learning-based testing* of *black-box systems* in *networked environments*. This sets the scope of this thesis as described in the following. By considering *black-box systems*, we assume to have minimal knowledge about systems. In particular, we assume no knowledge about their internal structure, but only require a way to interact with them. This means that a system only needs to provide an interface to interact with it and observe its reactions. By focusing on *testing*, we consider test interactions and observations of finite length, in contrast to other verification approaches such as model checking [46, 86]. As explained above, we consider *learning-based testing* as testing based on learned models. Therefore, it is also a part of *test-based learning*, a process involving interleaved testing of systems and learning of system models from test observations. In this process, we commonly iteratively refine our knowledge about black-box systems represented by learned automata models that we can use for model-based testing. We consider *networked environments*, such as the IoT. Since reliable communication is essential for the dependability of networked systems, we mostly concentrate on testing of communication protocols. However, we also consider aspects, such as the communication and interaction of software-based systems among each other and with the physical world. Especially testing the interaction with the physical world is becoming increasingly important in the context of the IoT, therefore we also investigate learning-based testing of so-called hybrid systems [196].

1.6.2 Research Context

The work that serves as basis for this thesis was performed in the project "Dependable Internet of Things in Adverse Environments"[2]. This project is a Lead project, an initiative funded by TU Graz to perform basic research on dependability in the IoT. At the time of writing, the project has entered the second project period after a successful evaluation of the first project period to which we contributed.

Researchers from the departments of Computer Science & Biomedical Engineering and Electrical & Information Engineering collaborate in this project to study various aspects of dependability. These aspects include reliability, availability, safety, confidentiality, and integrity. The project's goal is to provide methods and techniques to (1) increase dependability in these aspects and to (2) guarantee certain levels of dependability.

The first project period was organised into four subprojects. The work discussed in this thesis has been carried out in the subproject *Dependable Composition*. This subproject focused on developing techniques to ensure correct communication between IoT devices (things). By verifying communication to be correct, it shall be guaranteed that the composition of communicating things is dependable, given that these things fulfil additional criteria. An example of an additional criterion is that sensitive information should not be accessible to third parties, such as attackers. The subproject *Dependable Computing* worked on providing guarantees with respect to security.

1.6.3 Research Plan

The work discussed within this thesis can roughly be grouped into two phases: (1) an exploratory phase determining the state-of-the art in learning-based testing and identifying research questions related to the domain of networked environments, and (2) a focused phase addressing the identified research questions.

Exploratory Research

In the exploratory phase, we surveyed the literature on work combining model learning and testing and applied existing automata-learning techniques for model-based testing to gain practical experience.

We carried out the literature survey in collaboration with Wojciech Mostowski, Mohammad Reza Mousavi, and Masoumeh Taromirad from Halmstad University and specifically focused on work that explicitly targets testing, that is, we included work that either uses model-based testing for model learning, or model learning for model-based testing. The literature survey was published in the book on the

[2]Project website: `http://dependablethings.tugraz.at`, accessed on November 4, 2019

Dagstuhl seminar “Machine Learning for Dynamic Software Analysis” that was attended by Mohammad Reza Mousavi from Halmstad University [26].

In the practice-oriented line of work, we learned deterministic discrete-time finite-state models of broker implementations of the IoT protocol Message Queuing Telemetry Transport (MQTT) [73, 153]. Through equivalence checking on model-level, we performed implicit model-based differential testing between these implementations to discover specification violations. We presented the applied approach and the results of this case study on learning-based testing at the ICST 2017 [263]. The main goal of this work was to get a better understanding of learning-based testing in general and more specifically to answer the following two research questions.

- **RQ E.1** Is learning-based testing effective in networked environments such as the IoT?
- **RQ E.2** What are the shortcomings of existing automata-learning approaches in practical applications?

Findings from Exploratory Research

We found several specification violations of MQTT implementations used in practice, therefore we concluded that **RQ E.1** can be answered positively. Learning-based testing is an effective method for failure detection in our considered domain. Chapter 3 discusses this line of research in more detail and revisits the research questions related to the exploratory research. In the course of the work on learning-based testing MQTT implementations, we identified shortcomings in the following aspects leading to the definition of further research questions discussed below.

- **Runtime:** we found that automata learning generally has a high runtime, especially in networked environments due to the high response times of the systems under learning.
- **Non-deterministic & stochastic behaviour:** existing deterministic learning techniques fail when systems show non-deterministic or stochastic behaviour. More concretely, these techniques do not produce any tangible results in these situations.
- **Timing-related behaviour:** through the application of discrete-time learning, we had to ignore timing-related behaviour.

1.6.4 Problem Statements and Research Questions

Runtime. Initial learning experiments in our MQTT case study [263] took an excessive amount of time. There are two common strategies to counter such issues: (1) abstraction to learn system models with small abstract state space [1] and (2) smart test-case generation to reduce the number of test cases required for finding discrepancies between learned models and the system under learning [144]. We applied harsh abstraction focusing on selected properties of MQTT to enable learning. Despite this harsh abstraction, learning still took a large amount of time and due to this abstraction, we could not check certain properties. In multiple cases, it took several hours to learn models with less than twenty states. Moreover, we found that deterministic conformance testing algorithms do not scale, which is in line with findings by Howar et al. [144]. Consequently, we identified the following research questions.

- **RQ 1.1** Are randomised testing techniques a sensible choice for learning-based testing?
- **RQ 1.2** What guarantees can be given if randomised testing is applied?
- **RQ 1.3** Can learning with randomised conformance testing reliably generate correct system models?

- **RQ 1.4** Is fault-based testing, such as model-based mutation testing, applicable in automata learning?

Since deterministic conformance testing does not scale, randomised testing techniques should be investigated. The first two questions address the issue that random testing usually comes with weak or no guarantees. This carries over to models learned with random testing, that is, learned models may be incorrect. Thus, **RQ 1.2** could be rephrased as "How can potentially incorrect models help in verification and testing?". The third question asks whether complete deterministic conformance testing is necessary for learning correct automata. Testing is incomplete in general, but still it is very successful in verification and validation of software systems. Therefore, it should be investigated if similar observations can be made for conformance testing in learning. The last research question addresses a concrete form of test-case generation. It is motivated by our previous work in Professor Bernhard Aichernig's group, in which we researched and applied model-based mutation testing for fault-based testing of reactive systems [11, 19, 20, 21, 22, 23, 169]. In general, this form of testing injects known faults, so-called mutations, into a specification model and generates test cases that cover those faults. Model-based mutation testing was found to be especially effective in combination with random testing [22], therefore it is a natural choice for the black-box setting considered in learning.

Non-deterministic & Stochastic Behaviour. We were not able to finish all learning experiments in our MQTT case study, due to cases in which systems did not behave deterministically. Behaviour that is not deterministic is usually modelled as being non-deterministic or stochastic. Non-deterministic models may react to given inputs with responses from a finite set of choices, but they do not include information about the likelihood of each response, whereas stochastic models assign probabilities to the possible responses. There are also modelling formalisms incorporating non-deterministic and stochastic behaviour. To simplify our discussion, we refer to non-deterministic or stochastic behaviour as *uncertain behaviour*. Hence, we observe uncertain behaviour if the repeated application of the same input stimuli may produce different outputs.

Deterministic learning techniques, such as Angluin's L^* [37], usually do not produce any model if uncertain behaviour is observed during learning. Hence, it is necessary to apply alternative learning techniques that learn types of models which are able to capture uncertainties. There are learning approaches for non-deterministic [161, 283] and stochastic systems [199], however, the research in this area is in its early stages, thus we identified the following research questions.

- **RQ 2.1** Which modelling formalisms are appropriate in learning-based testing of uncertain behaviour?
- **RQ 2.2** When can we stop learning in the presence of uncertain behaviour?
- **RQ 2.3** Is test-based active learning feasible in the presence of stochastic behaviour?

In order to answer the first question, we need to investigate if uncertainties should be represented stochastically or non-deterministically. As noted above, systems involving uncertainties react with one of finitely many responses for a given input. This gives rise to the question of when we have seen all possible reactions during learning as those should be captured in the learned model. **RQ 2.2** addresses this issue and is also related to **RQ 1.2**. Due to the black-box nature of learning, we can never be entirely sure that we have seen every possible behaviour, but we may still be able to give guarantees relative to the model we learned. Finally, **RQ 2.3** is a specific question concerning active learning. In the literature, we found test-based active learning of non-deterministic models [161, 283] based on Angluin's L^* [37], but we did not find L^*-based learning of stochastic models applicable in a test-based setting. Therefore, we decided to examine whether L^*-based learning is feasible in this context and if it is feasible, how it differs from other approaches.

Timing-Related Behaviour. The MQTT specification allows users to set a *Keep Alive* time defining the maximum duration between consecutive packets sent by a client that must not be exceeded [73]. Similar timeout mechanisms are also available in other communication protocols like the transmission control protocol (TCP) [235]. Since discrete-time models can hardly capture such functionality, we had to ignore all timing-related behaviour in our MQTT case study. A literature survey revealed that learning of real-time system models is often restricted in expressiveness and difficult to implement via testing. Therefore, we decided to study alternative approaches and defined the following two research questions.

- **RQ 3.1** Is learning of real-time system models feasible through metaheuristic search-based techniques?
- **RQ 3.2** What assumptions are sufficient to enable learning in the context of real-time systems?

There are two major approaches to automata learning, namely passive state-merging-based learning and active L^*-based learning; see Section 1.5. Both, the passive [279] and the active [126] approach, have been studied for real-time systems. Motivated by the recent success of genetic programming [167] in program synthesis [159], we decided to investigate the feasibility of metaheuristic model learning for real-time systems. This investigation gave rise to the first research question **RQ 3.1**. Our goal was to overcome limitations of existing approaches with respect to expressiveness. In other words, we wanted to learn models covering a wider range of possible systems. The second research question **RQ 3.2** concerns learnability. It asks for properties that a system under learning needs to fulfil for learning to be feasible or successful.

1.6.5 Thesis Statement

Test-case generation based on intermediate learned automata models combined with randomisation enables active learning of accurate automata models.

Exhaustive conformance testing is usually infeasible in practice, therefore testing strategies that take the targeted application domain into account have to be devised. Promising scalable strategies include, but are not limited to, coverage-guided testing, reachability-directed testing, and fault-based testing. Appropriately chosen testing strategies enable accurate learning as well as guarantees with respect to verification objectives in a black-box setting.

1.7 Structure of this Thesis

This thesis is structured as follows. This chapter, Chapter 1, is the introductory chapter. It describes the motivation, the context and the outline of this thesis, defines research questions and discusses the work that forms the basis of this thesis. In Section 1.9, it introduces notational conventions that are used throughout this thesis.

The main part of this thesis is structured along similar categories as the research questions introduced above. Each chapter in the main part discusses learning-based testing or test-based learning of different types of systems. We cover model learning for deterministic systems, stochastic systems, real-time systems, and hybrid systems which extend real-time systems.

The first three chapters focus on learning of deterministic system models. Chapter 2 provides an introduction to test-based active automata learning which serves as background knowledge for the following chapters. Chapter 3 covers the practical exploratory research. It discusses learning-based testing of MQTT brokers. In Chapter 4, we concentrate on efficient conformance testing in active automata learning. We present an approach to fault-based test-case generation in automata learning in this chapter.

Additionally, we also present benchmarking experiments that evaluate active automata learning configurations with respect to efficiency.

Chapter 5 to Chapter 7 discuss our work with regard to systems involving uncertainties. Our work in this area focused on learning uncertain behaviour that is modelled stochastically and covers both learning-based testing and test-based learning. Chapter 5 introduces background knowledge on stochastic models and verification. Learning-based verification is the focus of Chapter 6. The chapter presents an approach to learning-based testing of stochastic systems. This approach learns near-optimal testing strategies with respect to reachability objectives. Test-based learning of stochastic system models is covered by Chapter 7, which presents an L^*-based learning approach for Markov decision processes (MDPs).

In Chapter 8, we discuss metaheuristic learning of models of real-time systems. The chapter introduces a passive learning framework which is extended to active learning in Chapter 9. Finally, Chapter 10 presents our work on learning behaviour models of hybrid systems.

The final two chapters revisit the topics and work covered in this thesis. We first discuss work related to the topics considered in this thesis in Chapter 11. The chapter concentrates on work in the intersection between automata learning and model-based testing, closely following our literature survey [26], but it also discusses other related work, such as strategy generation for stochastic systems. Chapter 12 concludes this thesis by first summarising the performed work. Subsequently, we discuss our findings with respect to the research questions defined in Section 1.6.4. Finally, we give an outlook on potential future work.

1.8 Contributions and Publications

1.8.1 Contributions

With our research on learning-based testing, we contributed to the state of the art in various ways. Our contributions span the development of learning and testing approaches, the implementation of these approaches, case studies on learning-based testing and test-based learning, and the creation of benchmark models for learning and conformance testing. In more detail, our contributions are as follows.

- *Learning-based testing of communication protocols [263]:* to study the applicability of automata learning for semi-automatic fault detection without prior knowledge, we performed a case study on learning-based testing of the IoT protocol MQTT. In the course of this work, we learned models that are now part of a set of benchmark models for automata learning and conformance testing [214].
- *Learning-based testing of stochastic systems [16, 18]:* we developed and implemented a learning-based testing approach for reachability objectives of stochastic systems. Our extensive evaluation showed that this testing approach generates near-optimal testing strategies.
- *Efficient test-based learning of deterministic systems [15, 17]:* with the goal of improving the efficiency of test-based learning, we developed a fault-based testing technique for active automata learning that derives test suites from learned intermediate models. For this purpose, we designed a mutation operator injecting faults into learned models that are specifically tailored towards active automata learning. We evaluated our implementation of this technique in combination with various learning algorithms by comparing it to alternative conformance testing techniques.
- *Test-based learning [29, 264, 265, 266, 267]:* we developed and implemented test-based learning methods for various types of system models including stochastic systems, timed systems, and hybrid systems. In each case, we applied or extended state-of-the-art techniques to enable learning of the targeted type of system. For learning of stochastic systems, we adapted and extended the

classic L^* algorithm [37]. To learn timed system models, we first developed a passive learning technique based on genetic programming [167] that was subsequently extended to an active setting in Andrea Pferscher's master's thesis [234], under co-supervision of the author of this thesis. Finally, we learned neural network models of hybrid systems by generating training data through learning-based testing.

- *Implementations:* the implementations of the developed learning and testing approaches are publicly available online in various forms. They are available in the form of source code [259, 260], as evaluation package including source code [261], and as demonstrator with graphical user interface [262].

1.8.2 Main Publications

The statutes of the Doctoral School of Computer Science at Graz University of Technology mandate that a dissertation needs to include an annotated list of publications explaining the relation between the dissertation and the publications. Additionally, joint work with third parties needs to be highlighted. The following list briefly introduces the content of each publication considered for this dissertation and the contributions of the author of this dissertation. It is sorted chronologically with respect to the date of creation and refers to publication venues. To ease presentation, contributions are described in first-person singular form.

This thesis builds upon research and findings presented in six peer-reviewed conference papers, two journal articles, one book chapter, one co-supervised master's thesis and one co-supervised bachelor's thesis.

1. ICST 2017 – *Model-Based Testing IoT Communication via Active Automata Learning* [263]: this conference paper discusses learning-based testing of the IoT protocol MQTT. Thus, it covers the practical part of our exploratory research performed for this thesis that we will discuss in Chapter 3.

   I presented the paper at the ICST 2017 and it was published in the proceedings of this conference. The initial idea for learning-based testing in the IoT was developed by my co-authors Bernhard Aichernig and Roderick Bloem. I designed the case study on MQTT and implemented the MQTT interface and the learning setup to perform the case study. Furthermore, I analysed the case study results and wrote the initial version of the paper. I polished the paper in collaboration with my co-authors and based on feedback from colleagues.

2. Dagstuhl Seminar – Machine Learning for Dynamic Software Analysis – *Model Learning and Model-Based Testing* [26]: this chapter is part of a book that has been published following a Dagstuhl Seminar on machine learning in the context of verification. The chapter presents a survey on works in the intersection between model learning and model-based testing. We wrote it in collaboration with researchers from Halmstad University. I wrote most of the section on learning. All authors contributed in approximately equal parts to the three main sections of the survey. I worked on this chapter in the exploratory phase of the research performed for this thesis. Chapter 11 uses content from the survey.

3. NFM 2017 – *Learning from Faults: Mutation Testing in Active Automata Learning* [15]: this conference paper presents an efficient fault-based conformance testing technique for active automata learning. It tackles issues related to learning runtime that we discovered in the exploratory research. This line of work is discussed in Chapter 4.

   I presented the paper at the NFM 2017 and it was published in the proceedings of this symposium. I developed, implemented, and evaluated the test-case generation approach including the mutation operator targeting the specifics of active automata learning. My supervisor Bernhard Aichernig helped to fine-tune certain aspects concerning mutation-based testing. I wrote the initial draft of the paper and polished it in collaboration with Bernhard Aichernig.

4. RV 2017 – *Probabilistic Black-Box Reachability Checking* [16]: this conference paper presents a technique for learning-based testing of stochastic systems with respect to reachability properties. The technique is a step towards learning-based verification of systems with uncertain behaviour via testing. Chapter 6 covers this technique.

   I presented the paper at the RV 2017 and it was published in the proceedings of this conference. I developed, implemented, and evaluated the presented online-testing approach. My supervisor Bernhard Aichernig provided help with respect to aspects, such as statistical model-checking. I wrote the initial draft of the paper and polished it in collaboration with Bernhard Aichernig.

5. JAR 2018 – *Efficient Active Automata Learning via Mutation Testing* [17]: following the presentation of our paper at the NFM 2017, we were invited to submit an extended version of our conference paper on efficient fault-based conformance testing in active automata learning [15] to the *Journal of Automated Reasoning* as part of the special issue on the NFM 2017. In addition to the content of the conference paper, the extended version includes optimisations and a substantially more thorough evaluation. These additions are also discussed in Chapter 4.

   I developed the optimisations of the testing technique and performed the extended evaluation. Moreover, I extended the conference paper by a discussion of these additions, an in-depth discussion of the applied mutation technique and several further improvements. I polished the article in collaboration with Bernhard Aichernig.

6. FMSD 2019 – *Probabilistic black-box reachability checking (extended version)* [18]: following the presentation of our paper at the RV 2017, we were invited to submit an extended version of our conference paper on learning-based testing of stochastic systems [16] to the journal *Formal Methods in System Design* as part of the special issue on the RV 2017. The additional material extending the original paper includes a heuristic stopping criterion for our testing technique and a more thorough evaluation. Chapter 6 covers these aspects as well.

   I developed the new stopping criterion and performed the extended evaluation. Additionally, I extended and improved the original conference paper in various other aspects, for instance, by an extensive introduction into learning-based testing and a discussion on convergence. I polished the article in collaboration with Bernhard Aichernig.

7. FORMATS 2019 – *Time to Learn – Learning Timed Automata from Tests* [267]: this conference paper presents a genetic-programming-based technique for learning models of real-time systems from test observations. With that, we tackle issues with respect to the expressiveness of modelling formalisms supported by existing learning techniques. Chapter 8 is based on this work.

   I presented the paper at the FORMATS 2019 and it was published in the proceedings of this conference. I developed, implemented, and evaluated the genetic-programming-based technique described in the conference paper. In collaboration with my co-authors, I extended and fine-tuned the technique as well as the experiments performed in the evaluation. I wrote the initial draft of the paper, before extending and polishing it in collaboration with my co-authors. Furthermore, Andrea Pferscher and I implemented a demonstrator including a graphical user interface. A technical report focusing on other aspects than the conference paper is available at `arxiv.org` [264].

8. ICTSS 2019 – *Learning a Behavior Model of Hybrid Systems Through Combining Model-Based Testing and Machine Learning* [29]: this conference paper discusses the application of model-based testing for training data generation in machine learning. It demonstrates various approaches for data generation in a platooning case study. Chapter 10 covers the work presented in this paper.

   I presented the paper at the ICTSS 2019 and it appeared in the proceedings of this conference. **The paper was selected as being the best paper of the ICTSS 2019.**

   I implemented the learning-based and the model-based testing techniques used for performing the testing part of the presented case study. In collaboration, we developed the idea of applying model-based testing to generate training data for machine learning. I wrote the testing-related parts of the conference paper and also contributed to other parts. An extended version of the conference paper is available as preprint [28].

Our initial efforts in this line of research have been presented at the "Lange Nacht der Forschung" 2018[3]. At this event, we presented learning-based testing using the platooning case study to the general public, focusing on testing for functional correctness.

9. FM 2019 – *L^*-Based Learning of Markov Decision Processes* [265]: this paper presents L^*-based learning of MDPs. The main contributions of the paper encompass development, evaluation, and implementation of a sampling-based active automata learning approach for MDPs that relies solely on test observations. This approach is discussed in Chapter 7.

   I presented the paper at the FM 2019 and it appeared in the proceedings of this conference. I developed and implemented the sampling-based learning algorithm, including auxiliary techniques, like model-based equivalence testing of MDPs. In collaboration with my supervisor Bernhard Aichernig, I fine-tuned various aspects of the learning algorithm. I wrote the initial version of the conference paper where an extended version of the paper is available as preprint [266]. In collaboration with my co-authors, I improved the presentation, worked out proofs on convergence, and generally polished the paper.

10. Master's thesis of Andrea Pferscher – *Active Model Learning of Timed Automata via Genetic Programming* [234]: Andrea Pferscher's master's thesis extends our passive genetic-programming-based learning technique for models of real-time systems to an active learning technique. This is done by interleaving learning and testing. The master's thesis forms the basis of Chapter 9.

    I co-supervised this thesis, providing advice on the passive learning technique, on active automata learning, and on the evaluation of timed-automata learning. I also proofread the thesis.

11. Bachelor's thesis of Felix Wallner (not yet submitted) – *Benchmarking Active Automata Learning Configurations* [288] : Felix Wallner evaluated combinations of active automata learning algorithms with conformance-testing techniques, including our fault-based conformance testing approach [15, 17], in his bachelor's thesis project. Section 4.5 discusses the evaluation and its results which is part of the chapter on efficient conformance testing in active automata learning. At the time of writing, Felix Wallner has not yet submitted the final bachelor's thesis.

    I co-supervised the thesis project, providing guidance on the application of active automata learning and the evaluation of active automata learning.

1.8.3 Related Publications

In the course of the project "Dependable Internet of Things in Adverse Environments", I participated in joint research related to this thesis. The following publications have been created in these efforts, but are not discussed in-depth.

1. S&P 2016 – *Learning Models of a Network Protocol using Neural Network Language Models* [10]: this poster presented by Tobias Schrank proposes to learn neural networks capturing the message flow of the TLS protocol. The main author of the poster was Tobias Schrank.
2. IJC 2017 – *Dependable Internet of Things for Networked Cars* [130]: this article presents the work in the project "Dependable Internet of Things in Adverse Environments". The main author of the article was Bernhard Großwindhager.
3. FMCAD 2018 – *Automata Learning for Symbolic Execution* [24]: this conference paper presents a combination of active automata learning and symbolic execution to test systems comprising black-box and white-box components. Active automata learning automatically generates models of black-box components. These models can be composed with the white-box components to enable symbolic execution of the composition. The main author of the paper was Masoud Ebrahimi.

[3] `https://www.langenachtderforschung.at`, accessed on November 4, 2019

Table 1.1: Notational conventions

Notation	Condition	Meaning
S^*	S is a non-empty set	arbitrary-length sequences of elements in S
S^l		sequences of elements in S with length l
$s \cdot s'$	$s, s' \in S^*$	concatenation of s and s'
ϵ		empty sequence $\epsilon \in S^*$
$\lvert s \rvert$		length of sequence s
$\lvert S \rvert$		cardinality of set S
e	$e \in S$	sequence $e \in S^*$ of length one (elements lifted to sequences)
$s[i]$	$s = s_1 \cdots s_n \in S^*$	$s[i]$ is the i^{th} element s_i ... one-based indexed access
$s[< i]$		$s[< i] = s_1 \cdots s_{<i}$ ... subsequence of s with indexes $j < i$
$s[\leq i]$		$s[\leq i] = s_1 \cdots s_{\leq i}$ ... subsequence of s with indexes $j \leq i$
$s[\geq i]$		$s[\geq i] = s_{\geq i} \cdots s_n$ ... subsequence of s with indexes $j \geq i$
$s[> i]$		$s[> i] = s_{>i} \cdots s_n$ ... subsequence of s with indexes $j > i$
$s \ll s'$		s is a prefix of s', i.e. $\exists\, t \in S^* : s \cdot t = s'$
$s \gg s'$		s is a suffix of s', i.e. $\exists\, t \in S^* : t \cdot s = s'$
$A \cdot B$	$A, B \subseteq S^*$	pair-wise concatenation: $A \cdot B = \{a \cdot b \mid a \in A, b \in B\}$
$\mathit{prefixes}(A)$		prefixes of sequences in A: $\{a' \in S^* \mid \exists\, a \in A : a' \ll a\}$
$\mathit{suffixes}(A)$		suffixes of sequences in A: $\{a' \in S^* \mid \exists\, a \in A : a' \gg a\}$
$\mathcal{P}(C)$	C is a set	power set of C
$\mathbb{P}(a)$	a is an event	probability of a
$\mathcal{T}(e)$	$\mathcal{T}$ is a multiset	multiplicity of e in $\mathcal{T}$

1.9 Notation

This thesis mainly discusses automata learning and automata-based verification, therefore we introduce notational conventions in Table 1.1 that we will use throughout this thesis. The left column introduces some notation, the middle column specifies conditions on variables used in the table and the right column describes the corresponding meaning of the notation. Note that conditions in Table 1.1 are cumulative, that is, conditions introduced in a row also hold in the rows below. In addition to that, we will introduce some terminology and auxiliary functions below.

Terminology

A set of sequences $A \subseteq S^*$ is prefix-closed, if it contains all prefixes of all sequences in A, that is, if $A = \mathit{prefixes}(A)$. Analogously, A is suffix-closed if it contains all suffixes of all sequences in A, that is, if $A = \mathit{suffixes}(A)$.

Auxiliary Functions

Throughout this thesis, we will use three pseudo-random functions to perform probabilistic choices, for instance, to decide whether test-case generation should be stopped. The function *coinFlip* implements a biased coin flip performing binary probabilistic choice. It is defined for $p \in [0, 1]$ by $\mathbb{P}(\mathit{coinFlip}(p) = \mathbf{true}) = p$ and $\mathbb{P}(\mathit{coinFlip}(p) = \mathbf{false}) = 1 - p$. The function *rSel* selects a single sample e from a set S according to a uniform distribution, that is, $\forall e \in S\colon \mathbb{P}(\mathit{rSel}(S) = e) = \frac{1}{|S|}$. The function *rSeq* takes a set S and a length bound $b \in \mathbb{N}$ to create a random sequence s of elements in S. The length $l \leq b$ of s is chosen uniformly from $[0\,..\,b]$ and each element of s is chosen via $\mathit{rSel}(S)$.

All three pseudo-random functions require to be initialised. The respective initialisation operations take a *seed*-value for a pseudo-random number generator as input and return an implementation of the corresponding pseudo-random function. We assume these initialisation operations to be called prior to the execution of procedures that apply those functions. To simplify presentation, we do not make these calls explicit.

2 Introduction to Active Learning of Deterministic System Models

Declaration of Sources

The presentation in this chapter partly follows the presentation of active automata learning in our article on efficient fault-based conformance testing in the Journal of Automated Reasoning [17].

The following chapters cover our work on learning models of deterministic systems. All of our work in this area focused on the practical application of active automata learning in the minimally adequate teacher (MAT) [37] framework via testing. Therefore, we will introduce background knowledge on this topic in this chapter. This includes:

- Mealy machines, the modelling formalism we mainly used for deterministic systems,
- Angluin's L^* algorithm and the MAT framework [37],
- extensions of L^* and other active automata learning algorithms,
- conformance testing based on Mealy machine models,
- and abstraction in automata learning.

2.1 Mealy Machines

Mealy machines are well-suited to model reactive systems, such as implementations of communication protocols, and they have successfully been used in contexts combining learning and some form of verification [97, 112, 200, 263]. Moreover, the Java-library LearnLib [152] provides mature and efficient algorithms for learning Mealy machines.

Basically, Mealy machines are finite-state automata with inputs and outputs. The execution of a Mealy machine starts in an initial state and by executing inputs it changes its state. Additionally, exactly one output is produced in response to each input. Formally, Mealy machines can be defined as follows.

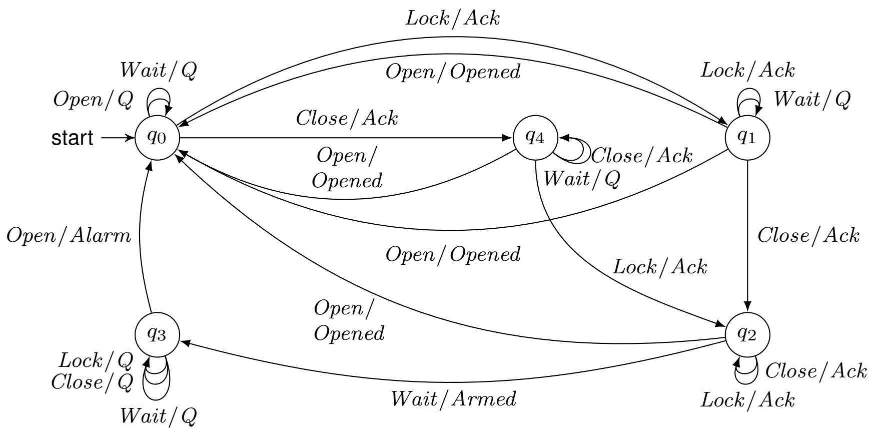


Figure 2.1: A Mealy machine modelling a car alarm system

Definition 2.1 (Mealy Machines).
A Mealy machine $\mathcal{M}$ is a 6*-tuple* $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ *where*

- Q *is a finite set of states,*
- q_0 *is the initial state,*
- I *is a finite set of input symbols,*
- O *is a finite set of output symbols,*
- $\delta : Q \times I \to Q$ *is the state transition function, and*
- $\lambda : Q \times I \to O$ *is the output function.*

We require Mealy machines to be input enabled and deterministic. Input enabledness demands that outputs and successor states must be defined for all inputs in all states, that is, δ and λ must be total. A Mealy machine is deterministic if it defines at most one output and one successor state for every pair of input and source state. This means that δ and λ must be functions in the mathematical sense. Example 2.1 shows a Mealy machine modelling a simple car alarm system. It is an adaptation of the car alarm system used in previous work by Bernhard Aichernig's group on test-case generation [22, 23].

Example 2.1 (Mealy Machine Model of a Car Alarm System). Figure 2.1 shows a Mealy-machine model of a simple car alarm system. We generally use this representation of Mealy machines, in which circles denote states and the edges denote transitions. Furthermore, each edge is labelled by a pair of an input and an output, separated by a slash. The initial state is marked by an edge without source state.

The car alarm system has the inputs Open, Close, and Lock for opening, closing, and locking the doors of a car. Another input Wait denotes waiting for some time. Depending on the current state, the system reacts with one of the outputs $O = \{\mathit{Ack}, \mathit{Alarm}, \mathit{Armed}, \mathit{Opened}, Q\}$ to each input, where Q denotes quiescent behaviour, i.e. the absence of outputs. There are two main requirements for the car alarm system: (1) it must be Armed if the doors have been closed and locked for some time and (2) it must produce an Alarm output if the doors are opened in the Armed state.

The second requirement is modelled by the transition from the state q_3 to q_0, which is formally described by $\delta(q_3, \mathit{Open}) = q_0$ and $\lambda(q_3, \mathit{Open}) = \mathit{Alarm}$. The state q_3 is the only state reached by a transition labelled with the output Armed.

We extend δ and λ to sequences of inputs in the standard way. Let $s \in I^*$ be an input sequence and $q \in Q$ be a state, then $\delta(q, s) = q' \in Q$ is the state reached by executing s starting in state q. This state is defined inductively by $\delta(q, \epsilon) = q$ and $\delta(\delta(q, s[1]), s[> 1])$ if $|s| > 0$. For $s \in I^*$ and $q \in Q$, the output function $\lambda(q, s) = t \in O^*$ returns the outputs produced in response to s executed in state q. Formally, $\lambda(q, \epsilon) = \epsilon$ and $\lambda(q, s) = \lambda(q, s[1]) \cdot \lambda(\delta(q, s[1]), s[> 1])$ if $|s| > 0$. Furthermore, let $\lambda(s) = \lambda(q_0, s)$ and $\delta(s) = \delta(q_0, s)$. For a state q, the set $acc(q) = \{s \in I^* \mid \delta(q_0, s) = q\}$ contains the access sequences of q. These sequences are the sequences leading to q. Note that certain works define a unique access sequence $s \in I^*$ for each q [151], but we are generally interested in a set of access sequences leading to a state.

We combine the state transition function and the output function to the transition relation $\rightarrow \subseteq Q \times I \times O \times Q$ and we write $q \xrightarrow{i/o} q'$, if $\delta(q, i) = q'$ and $\lambda(q, i) = o$. The transitive-reflexive closure $\rightarrow^* \subseteq Q \times I^* \times O^* \times Q$ of the transition relation is the smallest relation containing all $(q, \bar{i}, \bar{o}, q')$ such that $|\bar{i}| = |\bar{o}|$, $\delta(q, \bar{i}) = q'$, and $\lambda(q, \bar{i}) = \bar{o}$. In slight abuse of notation, we write $q \xrightarrow{\bar{i}/\bar{o}} q'$ for the transitive-reflexive closure, as for the one-step transition relation, if $(q, \bar{i}, \bar{o}, q') \in \rightarrow^*$.

In learning and conformance testing, we usually take a black-box view of systems. This means that we cannot observe the state structure of the considered systems. We can, however, observe which outputs they produce in response to inputs. Therefore, we define observations and observation equivalence similar to Aarts et al. [5]. Informally, an observation is an input/output sequence consisting of outputs produced by a Mealy machine in response to an input sequence. We also refer to observations as traces of Mealy machines.

Definition 2.2 (Observations).
Given a Mealy machine $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$, the observations of $\mathcal{M}$ in state q are $obs(q)_{\mathcal{M}} = \{(i, o) \in I^* \times O^* | \exists q' \in Q : q \xrightarrow{i,o} q'\}$ and the observations $obs_{\mathcal{M}}$ of $\mathcal{M}$ are the observations in the initial state, that is, $obs_{\mathcal{M}} = obs_{\mathcal{M}}(q_0)$.

Definition 2.3 (Observation Equivalence).
Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same input alphabets and the same output alphabets are observation equivalent, denoted $\mathcal{M}_1 \equiv \mathcal{M}_2$, iff $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

Definition 2.3 states that two Mealy machines are observation equivalent if they produce the same observations starting from their initial states. Since we usually take a black-box view, we mainly talk about observation equivalence. Therefore, we will generally refer to observation equivalence as equivalence in the context of Mealy machines. Equivalence of Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ can equivalently be defined with respect to their output functions λ_1 and λ_2. $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent iff $\forall i \in I^* : \lambda_1(i) = \lambda_2(i)$. In conformance testing, we want to find input sequences that show non-equivalence, therefore we define counterexamples to equivalence as follows.

Definition 2.4 (Counterexamples to Observation Equivalence).
Given two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ with output functions λ_1 and λ_2, a counterexample to observation equivalence between $\mathcal{M}_1$ and $\mathcal{M}_2$ is an input sequence $i \in I^*$ such that $\lambda_1(i) \neq \lambda_2(i)$.

2.2 L* and the Minimally Adequate Teacher Framework

We learn deterministic discrete-time automata in the minimally adequate teacher (MAT) framework introduced by Angluin for the L^* algorithm [37]. Algorithms in this framework interact with a so-called minimally adequate teacher (MAT) to learn automata accepting some unknown regular language or modelling a black-box SUL. This section presents the MAT framework and how algorithms in this framework usually work by abstractly describing L^*. Then, we will discuss learning of Mealy machines in more detail to give an intuition on how models of reactive systems can be learned in the MAT framework. Finally, we will discuss how L^*-based learning algorithms derive the structure of learned models from queried data.

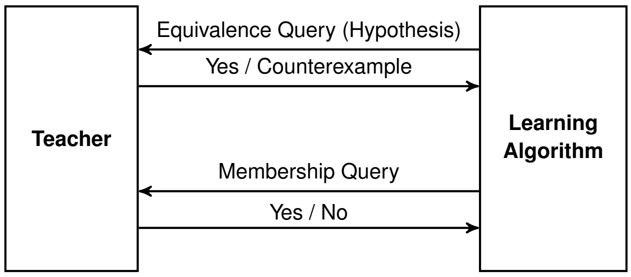


Figure 2.2: The interaction between a learning algorithm and a MAT [275]

2.2.1 Minimally Adequate Teacher Framework

A MAT usually needs to be able to provide answers to two types of queries that are posed by learning algorithms in the MAT framework. These two types of queries are commonly called *membership queries* and *equivalence queries*; see Figure 2.2 for a schematic depiction of the interaction between the learning algorithm, also called learner, and the MAT, also called teacher. In order to understand the basic notions of queries, consider that Angluin's original L^* algorithm is used to learn a DFA representing a regular language known to the teacher [37]. Given some alphabet, the L^* algorithm repeatedly selects strings and performs membership queries to check whether these strings are in the language to be learned. The teacher may answer with either *yes* or *no*.

After some queries, the learning algorithm uses the knowledge gained so far and forms a hypothesis. A hypothesis is a DFA consistent with the obtained information which is supposed to accept the regular language under consideration. The algorithm presents the hypothesis to the teacher and issues an equivalence query in order to check whether the language to be learned is equivalent to the language accepted by the hypothesis automaton. The response to this kind of query is either *yes*, signalling that a correct DFA has been learned, or a counterexample to equivalence. Such a counterexample serves as a witness showing that the learned model is not yet correct, that is, it is a word in the symmetric difference of the language under learning and the language accepted by the hypothesis.

After processing a counterexample, the learner starts a new *round* of learning. The new round again involves membership queries and a concluding equivalence query. This general mode of operation is basically used by all algorithms in the MAT framework with some adaptations. These adaptations may, for instance, enable the learning of Mealy machines as described in the following.

2.2.2 Learning Mealy Machines

Margaria et al. [200] and Niese [218] were among the first to learn Mealy-machine models of reactive systems using an L^*-based algorithm. Another L^*-based learning algorithm for Mealy machines has been presented by Shahbaz & Groz [246]. They reuse the structure of L^*, but substitute membership queries for *output queries*. Instead of checking whether a string is accepted, the learner provides inputs and the teacher responds with the corresponding outputs. For a more practical discussion, consider the instantiation of a teacher. Usually we want to learn the behaviour of a black-box SUL of which we only know the interface, that is, we only know the SUL's inputs and outputs. Hence, output queries are conceptually simple: provide inputs to the SUL and observe the produced outputs. In other words, an output query can be performed by executing a single test case and recording the test observations. However, there is a slight hidden difficulty. Similar to the original L^* algorithm [37], Shahbaz & Groz [246] assume that outputs are produced in response to inputs executed from the **initial** state. For this reason, we need to have some means to reset a system. Moreover, we generally cannot check for equivalence directly, because we take a black-box view of the SUL. It is thus necessary to approximate equivalence queries, for instance, via conformance testing as implemented in LearnLib [152], an automata learning library for Java. To summarise, a learning algorithm for Mealy machines relies on three operations:

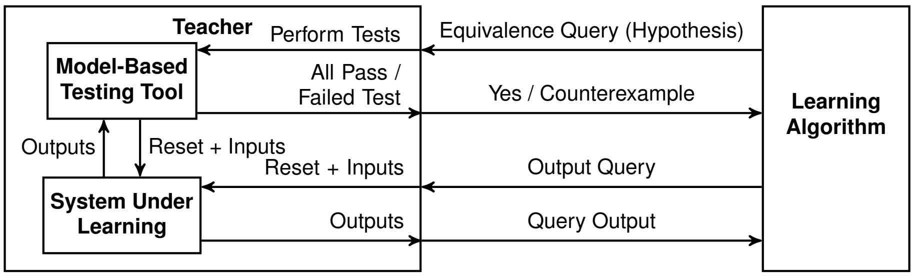


Figure 2.3: The interaction between a learning algorithm and a MAT communicating with a SUL to learn Mealy machines [275]

1. *reset:* resets the SUL
2. *output query:* performs a single test case executing inputs and recording outputs
3. *equivalence query:* tests for conformance between SUL and hypothesis, by generating and performing a set of test cases, often referred to as test queries. During each test-case execution, the teacher compares the outputs of the SUL with the outputs predicted by the current hypothesis.

As shown in Figure 2.3, the teacher is usually a component communicating with the SUL. It includes a model-based testing tool to generate test queries for equivalence queries. An equivalence query results in a positive answer if all generated test queries pass. A test query passes if the SUL produces the same outputs as the hypothesis, otherwise it fails. If there is a failing test query, the corresponding input sequence is returned as a counterexample to the learner.

Learning Process for Black-Box Systems

Based on the three operations described above, active automata learning of black-box SULs usually implements the general pattern given in pseudocode by Algorithm 2.1. We can see that this algorithm follows the active automata learning process introduced for L^*. It performs output queries until it can build

Algorithm 2.1 General pattern of active automata learning for black-box SULs

```
Input: SUL with a RESET operation and a function TEST : $I^* \to O^*$ executing a sequence of inputs and
    collecting the produced outputs
Output: learned Mealy machine
 1: repeat
 2:     $t \leftarrow$ select sequence for output query
 3:     RESET
 4:     $u \leftarrow$ TEST$(t)$                                  ▷ perform output query
 5:     STORE$(t, u)$
 6: until sufficient information to build hypothesis
 7: $\mathcal{H} = \langle Q_h, q_{0h}, I, O, \delta_h, \lambda_h \rangle \leftarrow$ build hypothesis
 8: Generate test suite $Ts \subset I^*$ from $\mathcal{H}$                ▷ perform equivalence query
 9: for all $t \in Ts$ do
10:     RESET
11:     $u \leftarrow$ TEST$(t)$
12:     if $u \neq \lambda_h(t)$ then
13:         extract and store information from counterexample $t$ and goto 1
14:     end if
15: end for
16: return $\mathcal{H}$
```

Table 2.1: Initial observation table for the car alarm system

	$S \backslash E$	$Open$	$Lock$	$Close$	$Wait$
S	ϵ	Q	Ack	Ack	Q
$S \cdot I$	$Lock$	$Opened$	Ack	Ack	Q
	$Open$	Q	Ack	Ack	Q
	$Close$	$Opened$	Ack	Ack	Q
	$Wait$	Q	Ack	Ack	Q

a hypothesis (Line 1 to Line 6). Then, it performs an equivalence query via testing (Line 8 to Line 15), and if it finds a counterexample to equivalence, it returns to performing output queries. Aside from this general process, there are also algorithm-dependent operations, which we left abstract in Algorithm 2.1, like how output queries are selected (Line 2) and how information is stored (Line 5).

Deriving Model Structure

We will now discuss how the structure of models is derived in active automata learning on the basis of the car alarm system introduced in Example 2.1 by performing one round of learning. For that, we execute Algorithm 2.1 until Line 13.

The way information is stored, what output queries are performed, and the amount of information necessary to build a hypothesis depends on the specific learning algorithm. A prefix-closed set of input sequences identifying hypothesis states is commonly maintained by various active automata learning approaches [37, 151, 258]. The states reached by these sequences are usually distinguished by outputs produced in response to another set of input sequences. In other words, states are distinguished by their future behaviour [258].

The following discussion is roughly based on the L^*_M algorithm for Mealy machines by Shahbaz and Groz [246] and Angluin's L^* [37]. For this discussion, let $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ be the Mealy machine modelling the car alarm system introduced in Example 2.1, serving as our SUL. L^* and its adaptations for Mealy machines commonly store information in observation tables which are triples $\langle S, E, T \rangle$, where $S \subset I^*$ is a prefix-closed set of input sequences identifying states, $E \subset I^*$ is a suffix-closed set of input sequences distinguishing states (with the exception that $\epsilon \notin E$), and $T : (S \cup S \cdot I) \cdot E \to O^+$ is a mapping storing the outputs of the SUL produced in response to some $e \in E$, after executing an input sequence $s \in S \cup S \cdot I$. Note that the sequences $S \cdot I$, the one-input extensions of S, identify transitions in learned models. Observation tables in L^*_M are initialised by setting $S = \{\epsilon\}$ and $E = I$.

Example 2.2 (Filling the Observation Table of the Car Alarm System). We first need to make output queries such that T is defined for all $(S \cup S \cdot I) \cdot E$. As we initially have $S = \{\epsilon\}$ and $E = I$, the first observation table is shown in Table 2.1. The upper part of the table rows contains the information for all sequences in S, that is, only ϵ, whereas the lower part contains the information for all sequences in $S \cdot I$. The columns are labelled by elements in E. For instance, the right-most cell in the bottom row stores $T(Wait \cdot Wait) = Q$, where the first $Wait$ is in $S \cdot I$ and the second $Wait$ is in E. The output Q (quiescent behaviour) is produced in response to the second $Wait$.

L^*-based approaches represent rows by a function $row(s) : E \to O^+$ for $s \in S \cup S \cdot I$ and consider each unique row in the observation to represent a different hypothesis state. In order to be able to build well-defined hypotheses from observation tables, L^* requires two conditions to hold. These conditions are called closedness and consistency and they are checked in Line 6 of Algorithm 2.1. Closedness requires for each unique row $row(s)$ that there must be an $s' \in S$ with $row(s) = row(s')$, thus the set S must cover all unique rows. This condition is required to ensure that transitions can be created for all states and all inputs. It can be seen that the observation table in Table 2.1 is not closed, as there is no $s' \in S$ such that $row(s') = row(Lock)$. Consistency requires that for all pairs of equivalent rows

Table 2.2: First closed and consistent observation table for the car alarm system

	S \ E	$Open$	$Lock$	$Close$	$Wait$
S	ϵ	Q	Ack	Ack	Q
	$Lock$	$Opened$	Ack	Ack	Q
	$Lock \cdot Close$	$Opened$	Ack	Ack	$Armed$
	$Lock \cdot Close \cdot Wait$	$Alarm$	Q	Q	Q
$S \cdot I$	$Open$	Q	Ack	Ack	Q
	$Close$	$Opened$	Ack	Ack	Q
	$Wait$	Q	Ack	Ack	Q
	$Lock \cdot Open$	Q	Ack	Ack	Q
	$Lock \cdot Lock$	$Opened$	Ack	Ack	Q
	$Lock \cdot Wait$	$Opened$	Ack	Ack	Q
	$Lock \cdot Close \cdot Open$	Q	Ack	Ack	Q
	$Lock \cdot Close \cdot Lock$	$Opened$	Ack	Ack	$Armed$
	$Lock \cdot Close \cdot Close$	$Opened$	Ack	Ack	$Armed$
	$Lock \cdot Close \cdot Wait \cdot Open$	Q	Ack	Ack	Q
	...	...	...	...	...

labelled by S, their one-input extensions must be equivalent too, that is, for $s, s' \in S$ if $row(s) = row(s')$ then $row(s \cdot i) = row(s' \cdot i)$ for all $i \in I$. This condition is required to build deterministic hypotheses. Table 2.1 is obviously consistent. Consistency violations are resolved by adding new columns to E. The following example demonstrates how to make our initial observation table closed and consistent.

Example 2.3 (Creating a Closed and Consistent Observation Table). Table 2.1 is not closed, therefore we add $Lock$ to S to make the table closed. We could as well have added $Close$. This requires us to issue additional output queries. In this process, we find additional closedness violations that we fix by adding $Lock \cdot Close$ and $Lock \cdot Close \cdot Wait$ to S as well. Table 2.2 shows the first closed and consistent observation table created during learning of the car alarm system. We omitted some rows labelled by $S \cup I$ for presentation purposes. Note that we did not extend E, as we did not find consistency violations.

Given a closed and consistent observation table, we can create a hypothesis by creating a state for each unique row $row(s)$. The rationale behind this is based on the Myhill-Nerode theorem for formal languages [216]. For further information on the Myhill-Nerode theorem and an adaptation to Mealy machines, we refer to Steffen et al. [258]. The transition function δ_h of hypotheses is defined by $\delta_h(row(s), i) = row(s \cdot i)$ and the output function λ_h is given by $\lambda_h(row(s), i) = T(s \cdot i)$, where $s \in S$. Finally, the initial state is $row(\epsilon)$. This construction is well-defined due to closedness, consistency, $\epsilon \in S$, and $I \subseteq E$. An important property of this construction is that automata learned via L_M^* have the minimal number of states among all Mealy machines consistent with the information stored in the observation tables [246]. Consequently, learned hypotheses converge to a Mealy machine equivalent to the SUL with the least number of states possible.[1] This property is generally shared by other active automata learning algorithms in the MAT framework as well [258].

Our next example shows the hypothesis derived from Table 2.2.

Example 2.4 (Deriving a First Hypothesis with $\mathbf{L_M^*}$). After the first batch of output queries, we derive the hypothesis shown in Figure 2.4 from the closed and consistent observation table presented in Table 2.2. The set S contains access sequences of all four states of the hypothesis. The sequences in E distinguish them and the one-input extensions $S \cdot I$ define transitions in the hypothesis. We also see that sequences with equivalent row functions are access sequences for the

[1]The final hypothesis is usually not uniquely defined, but it is equivalent to the smallest possible automaton up to isomorphism [37].

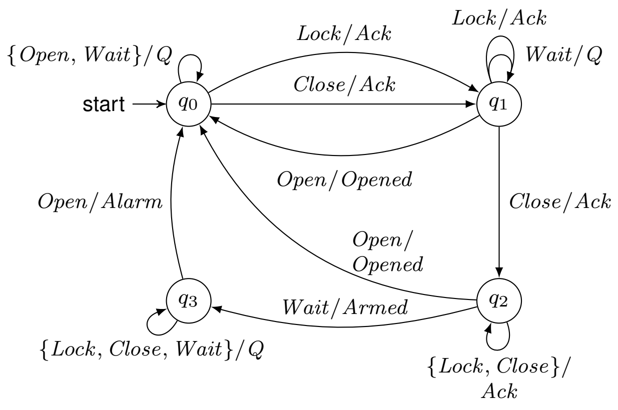


Figure 2.4: A first hypothesis of the car alarm system derived during learning

same state. Consider the sequences ϵ and $Open$ for instance, it holds that $row(\epsilon) = row(Open)$ and both sequences lead to the same state in the hypothesis.

To conclude the first round, we shall perform an equivalence query via testing. The following example shows a counterexample that we may find during testing. There are various strategies to process counterexamples in order to extract information from them. The original L^* algorithm, for instance, adds all prefixes of a counterexample to S [37], while another strategy proposed by Maler and Pnueli adds all suffixes of a counterexample to E [195], except for the empty sequence. We will further investigate counterexample processing further in the next section.

Example 2.5 (A Counterexample for the Car Alarm System). Suppose that we generated a set of test cases including the input sequence $t = Close \cdot Lock \cdot Wait$ in Line 8 of Algorithm 2.1. The hypothesis produces the outputs $Ack \cdot Ack \cdot Q$ for t, while the SUL produces $Ack \cdot Ack \cdot Armed$ (see Figure 2.1). Hence, the input sequence t is a counterexample to equivalence which we can use to improve our hypothesis.

2.3 Improvements in Active Automata Learning

Angluin provided the basis for active automata learning by introducing the MAT framework [37]. The runtime complexity and the space complexity of the L^* algorithm are polynomial in the size of the learned automata and the length of the longest counterexample returned from an equivalence query, but neither is optimal. In the following, we will discuss some improvements of active automata learning with regard to efficiency.

2.3.1 Reduced Observation Tables & Distinguishing Suffixes

Rivest and Schapire [240] proposed two influential improvements of Angluin's L^* [37] which are related to each other. We will refer to the L^* algorithm with these improvements as RS algorithm. Recall that an observation table is a triple $\langle S, E, T\rangle$ and that L^* requires consistency to derive hypotheses. The first improvement of the RS algorithm is that it maintains reduced observation tables. These observation tables satisfy the condition $\forall s, s' \in S : s \neq s' \rightarrow row(s) \neq row(s')$ which implies consistency. This condition ensures that each element of S corresponds to exactly one state in a hypothesis, thus $|S|$ cannot exceed the size of the minimal automaton modelling the SUL. The RS algorithm achieves this through advanced counterexample processing which is its second improvement. Rather than adding all prefixes of a counterexample c returned from an equivalence query to S, the RS algorithm extracts a single distinguishing suffix v from c and adds that to E. Through these improvements, the observation table size does not depend on the counterexample length which may significantly improve runtime.

A distinguishing suffix is a sequence that is able to distinguish the rows labelled by two sequences in $S \cup S \cdot I$. The key insight into the proposed technique is that the counterexample c can be split into three parts: $c = s' \cdot i \cdot v$, with $s' \in I^*$, $i \in I$, and $v \in I^*$. For such a splitting, there exists a uniquely defined $s \in S$ such that $\mathit{row}(s) = \mathit{row}(s' \cdot i)$ [240]. Furthermore, if c is a counterexample, then there exists a splitting such that executing $s \cdot v$ and $s' \cdot i \cdot v$ on the SUL produces different outputs. Note that it suffices to check the last output produced by each sequence [258]. Hence, adding v to E causes $\mathit{row}(s) \neq \mathit{row}(s' \cdot i)$. Making the observation table closed after adding v to E adds at least one sequence to S as well, thus creating a new hypothesis state. As a result, v basically splits the states reached by s and $s' \cdot i$ in the next hypothesis. Thus, we learn that s and $s' \cdot i$ reach different states in the SUL. This also serves as the basis for our fault-based testing technique presented in Chapter 4 [15, 17].

An appropriate splitting can be found via binary search using $\log(|c|)$ output queries. Hence, the counterexample processing does not cause a large runtime overhead. Without going into detail, we want to note that directly applying this counterexample processing strategy may lead to issues regarding canonicity of hypotheses. We refer to Steffen et al. [258] for more information. The following example demonstrates the counterexample processing.

Example 2.6 (Distinguishing Suffix Extraction). Figure 2.4 shows the first hypothesis learned for our car alarm system depicted in Figure 2.1 which is not yet correct. In Example 2.5, we presented $t = \mathit{Close} \cdot \mathit{Lock} \cdot \mathit{Wait}$ as a counterexample to equivalence between the hypothesis and the true car alarm system. Though not intended, the observation table in Table 2.2 satisfies the condition that all rows labelled by S are different.

We can split t into $s' = \epsilon$, $i = \mathit{Close}$ and $v = \mathit{Lock} \cdot \mathit{Wait}$. From Table 2.2, we derive $s = \mathit{Lock}$, as $\mathit{row}(\epsilon \cdot \mathit{Close}) = \mathit{row}(\mathit{Lock})$. Executing $\mathit{Close} \cdot \mathit{Lock} \cdot \mathit{Wait}$ and $\mathit{Lock} \cdot \mathit{Lock} \cdot \mathit{Wait}$ on the SUL shown in Figure 2.1 produces $\mathit{Ack} \cdot \mathit{Ack} \cdot \mathit{Armed}$ and $\mathit{Ack} \cdot \mathit{Ack} \cdot Q$, respectively. Hence, $v = \mathit{Lock} \cdot \mathit{Wait}$ is a suffix that is able to distinguish the states reached by Lock and Close. By adding v to E, we learn that Lock and Close reach different states in the SUL. Generally, we would perform a binary search for the values s', i, and v.

2.3.2 Tree-Based Storage

Kearns and Vazirani presented an active automata learning algorithm [160], which we abbreviate as KV algorithm, that uses trees to store the access sequences S and the distinguishing sequences E. The main idea behind the data storage used in the KV algorithm is that for each pair of sequences s, s' in S there exists an e in E distinguishing s from s'. Hence, this algorithm also maintains the property $|S| \leq n$, where n is the minimal number of states of the hypothesis.

For the sake of simplicity we assume that we learn DFA. In this case, a binary tree serves as data structure with leaves labelled by elements of S and nodes labelled by elements of E. Due to the one-to-one correspondence between S and the leaves, each leaf represents a hypothesis state. Therefore, each leaf also represents an equivalence class of access sequences. The exact leaf position of some sequence s, that is, the hypothesis state reached by s, is determined by an operation called *sifting* [160]. Starting at the root node, sifting determines the node label e in E and checks whether $s \cdot e$ should be accepted. If it should be accepted, we move to the right child and otherwise to the left child. In this way, sifting traverses the tree until reaching a leaf position. Hypotheses are constructed by creating a state for each s in S and creating transitions through sifting s' in $S \cdot I$.

Reducing Redundancy. A weakness of the KV algorithm is its counterexample processing. In its original version [160], the KV algorithm analyses counterexamples to determine two sequences s and s' corresponding to the same leaf in the current tree, but leading to different states in the SUL. It then splits the leaf labelled s, adds the counterexample prefix s' to S and adds the corresponding counterexample suffix e to E which distinguishes s and s'.

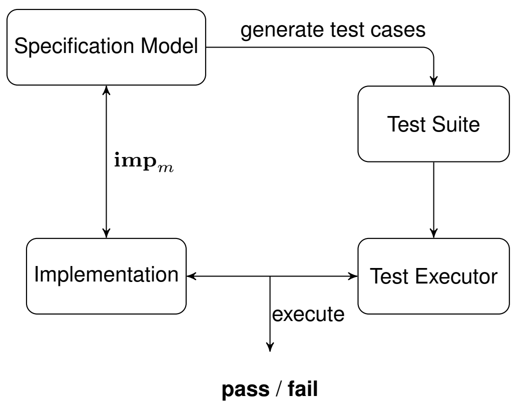


Figure 2.5: An overview of the conformance testing process [269]

Isberner et al. [151] note that this may introduce redundancies, especially when dealing with long counterexamples. Roughly speaking, a subsequence of s' may also lead to a new state and a subsequence of e may also be able to distinguish states. Therefore, they proposed the TTT algorithm [151], which uses trees for storage as well. In addition to the tree used by the KV algorithm, they keep track of a spanning tree formed by S in the hypothesis and a trie for storing all distinguishing suffixes E.

The TTT algorithm decreases the length of both prefixes and suffixes derived from counterexamples, whereby counterexample processing is partly based on the approach of Rivest and Schapire [240]. They also conduct the splitting of states and leaves due to counterexamples explicitly, but for more information we refer to the original publication [151]. At the time of writing, the TTT algorithm is presumably the most sophisticated algorithm for learning deterministic models, with optimal space complexity and best-known worst-case runtime complexity. The RS algorithm [240] has the same worst-case runtime complexity, though.

2.4 Conformance Testing

As explained in Section 2.2, we assume the existence of a teacher in active automata learning of Mealy machines. This teacher needs to be capable of answering two types of queries: output queries and equivalence queries. When we learn models of black-box systems, the former query can usually be answered through a single test, executing given inputs and observing the corresponding outputs. The latter query is more difficult to implement. Mostly, it is implemented through conformance testing as we discussed in our survey for instance [26]. We also outlined this in Line 8 of Algorithm 2.1, which generates a set of test cases to be executed on the SUL. In the following, we will therefore introduce the conformance testing problem for Mealy machines in active automata learning, discuss approaches to conformance testing, and conclude with considerations specific to conformance testing in learning.

Overview. Figure 2.5 provides an overview of the conformance testing process. In conformance testing, a specification model and an implementation are related through a conformance relation, denoted $\mathbf{imp}_m$ here. The first step in the testing process generates a set of test cases from the specification model, called test suite. Therefore, this kind of testing is also called *model-based testing*. A test executor, also called test driver, then executes these tests on the implementation to find faulty behaviour. Such faulty behaviour is a discrepancy in the behaviour described by the specification model and the implementation. In conformance testing, discrepancies are captured as violations of a conformance relation $\mathbf{imp}_m$.

2.4.1 The Conformance Testing Problem

In conformance testing based on finite state machines, we are generally given a state-machine model, also called specification, and our goal is to test whether a black-box implementation, also called system under test (SUT), conforms to the given model [184]. In this scenario, we assume that the model is correct and we want to determine whether the implementation behaves correctly with respect to the model. This view on conformance testing changes slightly in automata learning. Since we want to learn a faithful model of an implementation, we assume the implementation to be correct for the purpose of conformance testing and we test whether a learned hypothesis model conforms to the behaviour of the implementation. We will see in the following that this change of perspective does not greatly affect testing based on Mealy machines.

Conformance Relations

We formalise the testing problem by defining a conformance relation $\mathbf{imp}_m$, also called implementation relation, that expresses the conditions for an implementation to conform to a specification model [269]. The subscript m stands for *Mealy machine* in our case. For this purpose, it is usually assumed that the black-box implementation can be modelled in the formalism for which the relation $\mathbf{imp}_m$ is defined. This assumption is commonly referred to as *testing hypothesis* [269]. It is actually also present in active automata learning. If we use a learning algorithm for Mealy machines, we implicitly assume that our SUL can be modelled as a Mealy machine. Note that this hypothesis does not require an explicit implementation model, but just the possibility to express the dynamics of the implementation in the chosen modelling formalism.

Since we implement equivalence queries through conformance testing, we use observation equivalence as defined in Definition 2.3 as conformance relation. Therefore, we refer to the conformance relation also as equivalence relation. Given an *implementation Mealy machine* $\mathcal{I}$ and a *specification Mealy machine* $\mathcal{S}$, we say that $\mathcal{I}$ conforms to $\mathcal{S}$, denoted $\mathcal{I}\,\mathbf{imp}_m\,\mathcal{S}$, if and only if $\mathit{obs}_\mathcal{I} = \mathit{obs}_\mathcal{S}$.

Test-Case Generation

Having defined what a conformance relation is and how we instantiate conformance relations for testing Mealy machines, we will now formalise test-case generation and cite desirable properties of test-case generation. Our discussion of test-case generation follows Tretmans [271]. In the following, we assume an implementation Mealy machine $\mathcal{I}$ with output function λ_I and a specification Mealy machine $\mathcal{S}$ with output function λ_S as given. Furthermore, let M be the set of all Mealy machines and let *TEST* be the set of all test cases.

Test cases basically describe what input stimuli need to be provided to a SUT during test-case execution. A set of test cases is called a test suite. To determine the outcome of performing all test cases in a test suite, we need a procedure $\mathit{exec} : \mathit{TEST} \to \{\mathbf{fail}, \mathbf{pass}\}$. This procedure assigns a verdict $\mathit{exec}(t)$ of either $\mathbf{fail}$ or $\mathbf{pass}$ to the execution of a test case t, where $\mathbf{pass}$ denotes conformance between $\mathcal{I}$ and $\mathcal{S}$ with respect to t and $\mathbf{fail}$ denotes a conformance violation. A test case t with $\mathbf{fail}$ verdict thus demonstrates a discrepancy between $\mathcal{S}$ and $\mathcal{I}$. Due to input-enabledness and deterministic behaviour of Mealy machines, we use input sequences as test cases, thus we instantiate $\mathit{TEST} = I^*$. Since we test for observation equivalence, we assign verdicts based on outputs. Formally,

$$\mathit{exec}(t) = \begin{cases} \mathbf{pass} & \text{if } \lambda_I(t) = \lambda_S(t) \\ \mathbf{fail} & \text{otherwise.} \end{cases}$$

Note that test-case execution implicitly resets the implementation, because $\lambda_I(t)$ and $\lambda_S(t)$ return outputs produced in response to t starting from the initial state. As shorthand notations, we define a predicate denoting whether an implementation $\mathcal{I}$ passes a test case t and another predicate denoting whether $\mathcal{I}$ passes a test suite T: $\mathcal{I}\,\mathbf{passes}\,t \Leftrightarrow \mathit{exec}(t) = \mathbf{pass}$ and $\mathcal{I}\,\mathbf{passes}\,T \Leftrightarrow \forall t \in T : \mathcal{I}\,\mathbf{passes}\,t$.

Following Tretmans [271], we view test-case generation algorithms as functions $gen_{\mathbf{imp}_m} : M \to \mathcal{P}(\mathit{TEST})$, that is, functions mapping from Mealy machines to test suites. Ideally, it should hold that:

$$\forall \mathcal{I} \in M : \mathcal{I}\, \mathbf{imp}_m\, \mathcal{S} \Leftrightarrow \mathcal{I}\, \mathbf{passes}\, gen_{\mathbf{imp}_m}(\mathcal{S}) \tag{2.1}$$

The test-case generation function $gen_{\mathbf{imp}_m}$ should produce test suites T from $\mathcal{S}$ such that an implementation $\mathcal{I}$ passes T if and only if $\mathcal{I}$ conforms to $\mathcal{S}$. Such test suites are called *complete* for $\mathcal{S}$. However, completeness usually cannot be achieved in practice when testing black-box implementations. Unless we place restrictions on possible implementations, the set of implementations is infinitely large. Therefore, we place weaker requirements on test suites. We require test suites to be *sound*. A test suite T is sound if any conforming implementation passes it, that is, it corresponds to the left-to-right implication of Equation (2.1). By the contrapositive of this implication, any implementation not passing a sound test suite does not conform to the specification. A test suite satisfying the right-to-left implication is called *exhaustive* and can detect errors in all implementations that do not conform to the specification. A test-case generation algorithm producing complete test suites for any specification is called complete. The terms *sound* and *exhaustive* are analogously assigned to test-case generation algorithms.

In the following, we will discuss some basic test-case generation algorithms for Mealy machines. Hence, these algorithms produce sets of input sequences to be executed on the SUT. We generally assign verdicts via *exec*, therefore our test suites are sound. To see why this holds, recall from Section 2.1 that $\mathcal{I}\, \mathbf{imp}_m\, \mathcal{S}$ (i.e. $\mathcal{I} \equiv \mathcal{S}$) iff $\forall i \in I^* : \lambda_I(i) = \lambda_S(i)$. This implies $\forall i \in T : \lambda_I(i) = \lambda_S(i) \Leftrightarrow \mathcal{I}\, \mathbf{passes}\, T$ for any $T \subseteq I^*$.

2.4.2 Conformance Testing Approaches

We mainly use the Java library LearnLib [152] for learning Mealy machines, therefore we concentrate on testing algorithms available in LearnLib[2] at the time of writing this thesis. However, note that these algorithms are standard algorithms for testing Mealy machines. All testing algorithms generate test cases based on a specification $\mathcal{S} = \langle Q, q_0, I, O, \delta, \lambda \rangle$.

W-Method

The W-method by Vasilevskii [277] and Chow [83] is a well-known, early test-case generation technique. Originally, it was proposed as a technique for generating tests from a possibly incorrect design of a program's control structure in order to test the program with respect to a specification. Here, we will use it to generate test cases from a specification Mealy machine that also serves as specification in order to execute the generated test cases on an implementation. It requires an estimation m of the maximum number of states of the implementation and produces an exhaustive test suite under the assumption that the implementation has less than or equal to m states. In other words, if the implementation passes the produced test suite, then the implementation either conforms to the specification model or it has strictly more than m states.

Algorithm 2.2 shows pseudocode for test-case generation with the W-method. It applies concepts from finite-state machine theory. The set W is a set of characterising input sequences for the specification $\mathcal{S}$. This means that for any pair of states of $\mathcal{S}$ there is a sequence in W that can distinguish the states in the pair. Formally, we require that $\forall q, q' \in Q : q \neq q' \implies \exists w \in W : \lambda(q, w) \neq \lambda(q', w)$. For standard approaches to compute W, we refer to Gill [120], as Chow [83]. Furthermore, we create a set of input sequences, called *transitionCover* in Algorithm 2.2, that covers every transition of the specification at least once. Sequences in the concatenation of *transitionCover* and W are able to detect all errors in λ and δ in Mealy machines with at most the same number of states as $\mathcal{S}$. These errors are called operation errors and transfer errors by Chow [83]. The infix sequences in *middle* are necessary to detect

[2]See also `https://learnlib.de/` and `https://github.com/Learnlib/learnlib`, accessed on November 4, 2019

Algorithm 2.2 Test-case generation with the W-method [83]

```
Input: $\mathcal{S} = \langle Q, q_0, I, O, \delta, \lambda \rangle, m$
Output: test suite $T$
1: $stateCover \leftarrow \{seq \mid q \in Q, seq \in acc(q)\}$
2: $transitionCover \leftarrow stateCover \cdot I$
3: $middle \leftarrow \bigcup_{d=0}^{m-|Q|} I^d$
4: $W \leftarrow buildCharacterising(\mathcal{S})$
5: $T \leftarrow transitionCover \cdot middle \cdot W$
```

Algorithm 2.3 Random-words-based test-case generation

```
Input: $n_{\text{test}} \in \mathbb{N}$, $l_{\min} \in \mathbb{N}$, $l_{\max} \in \mathbb{N}$
Output: test suite $T$
1: $T \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $n_{\text{test}}$ do
3:     $l \leftarrow rSel([l_{\min} \mathinner{..} l_{\max}])$
4:     $test \leftarrow \epsilon$
5:     for $i \leftarrow 1$ to $l$ do
6:         $test \leftarrow test \cdot rSel(I)$
7:     end for
8:     $T \leftarrow T \cup \{test\}$
9: end for
```

missing or additional states. Altogether, every test case consists of three parts, a sequence to a transition, a potentially empty middle part, and a characterising sequence.

The partial W-method, or Wp-method, by Fujiwara et al. [117] improves the W-method by creating smaller test suites, while achieving the same guarantees. Both testing techniques are implemented in LearnLib [152] and take a depth parameter d specifying the maximum difference between the number of states of the specification and the implementation, expressed as $m - |Q|$ in Algorithm 2.2. The W-method and the Wp-method suffer from the same drawbacks. First, we usually do not know an upper bound m on the number of states of a black-box implementation. Second, the size of the **test suites** produced by both techniques **are exponential in** $\mathbf{d = m - |Q|}$, therefore it is usually not feasible to choose d conservatively, i.e., to choose large d.

Random Testing

LearnLib [152] implements the two completely random testing algorithms *random-words* and *random-walks*. These algorithms ignore the specification $\mathcal{S}$ to produce entirely random input sequences and they mainly differ in how the length of random sequences is chosen. Algorithm 2.3 and Algorithm 2.4 implement these testing algorithms; see Section 1.9 for the definitions of *coinFlip* and *rSel*. To simplify presentation, we present a slight modification of the random-walks algorithm. LearnLib does not specify a number of tests, but it specifies a limit on the overall combined length of tests. Moreover, LearnLib implements random walks as online testing. This form of testing basically combines test-case generation and execution in a single operation [278], testing the SUT and simulating the model in parallel. By doing that, LearnLib stops the test execution as soon as an error is detected. The online and the offline version perform the same steps, in case no error is detected.

Algorithm 2.3 takes a maximum number of test cases n_{test}, a minimum test-case length $l_{\min}$, and a maximum test-case length $l_{\max}$, whereas Algorithm 2.4 controls test-case length through a stopping probability p_{stop}. Hence, the length of test cases produced by Algorithm 2.4 follows a geometric distribution with parameter p_{stop}. In practice, both algorithms can be assumed to behave similarly as long as the maximum test-case length $l_{\max}$ is large enough in relation to the length of potential counterexamples.

Algorithm 2.4 Random-walks-based test-case generation

```
Input: $n_{\text{test}} \in \mathbb{N}, p_{\text{stop}} \in (0, 1)$
Output: test suite $T$
1: $T \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $n_{\text{test}}$ do
3:     $test \leftarrow \epsilon$
4:     repeat
5:         $test \leftarrow test \cdot rSel(I)$
6:     until $coinFlip(p_{\text{stop}})$
7:     $T \leftarrow T \cup \{test\}$
8: end for
```

In theory, random-walks is slightly favourable as it actually produces exhaustive test suites in the limit by letting n_{test} go to infinity. Because of the geometric length distribution, any test-case length and thereby any input sequence has a non-zero probability to be produced.

Random W-method. At the time of writing, the latest version of LearnLib [152] is 0.14.0 which implements randomised versions of the W-method and the Wp-method. Like test cases produced by the deterministic W(p)-method, each test case produced by the random W(p)-method consists of three parts: a sequence t leading to a transition, a middle sequence m and a characterising sequence c. Basically, t and c are chosen uniformly at random from their respective sets in the deterministic test-case generation counterpart. The middle sequence m is a random walk with a minimum length, that is, its length is geometrically distributed with some offset.

Compared to purely random testing, the random W(p)-method applies a notion of randomised coverage of transitions, by selecting the test-case prefix t such that some transition is covered (i.e. executed). Through this selection and through the selection of a characterising suffix c, the randomised W(p)-method benefits from incrementally extended knowledge about the SUT in the form of increasingly more accurate hypotheses. In contrast to the deterministic W(p)-method it scales well, as it produces as many test cases as requested, but it generally cannot provide the same guarantees.

2.4.3 Conformance Testing in Learning

The conformance testing approaches discussed above mainly focus on techniques that have been applied in active automata learning. For more information on testing based on finite state machines, such as testing with unique input/output sequences [241], we refer to a survey by Lee and Yannakakis [184]. In the following, we will discuss some theoretical and practical considerations regarding the application of conformance testing in learning.

Conformance Relation

In conformance testing, a conformance relation $\mathbf{imp}_m$ provides the theoretical basis to decide if an implementation $\mathcal{I}$ conforms to a specification $\mathcal{S}$. We test if $\mathcal{I}\,\mathbf{imp}_m\,\mathcal{S}$ with a finite test suite T, where $\mathcal{I}$ is a black-box system and $\mathcal{S}$ is a given Mealy machine. This is reversed in learning, that is, we are given a hypothesis Mealy machine $\mathcal{H}$ and want to check if its behaviour conforms to $\mathcal{I}$. Since $\mathbf{imp}_m$ is an equivalence relation, we have $\mathcal{I}\,\mathbf{imp}_m\,\mathcal{H} \Leftrightarrow \mathcal{H}\,\mathbf{imp}_m\,\mathcal{I}$, therefore testing for $\mathcal{H}\,\mathbf{imp}_m\,\mathcal{I}$ is justified. More concretely, a test case shows a violation of the conformance relation $\mathcal{I}\,\mathbf{imp}_m\,\mathcal{H}$ iff it shows a violation of $\mathcal{H}\,\mathbf{imp}_m\,\mathcal{I}$. This does not hold in general, for instance, the input-output conformance (**ioco**) relation is not reflexive [270]. Now that we have established that testing with respect to $\mathbf{imp}_m$ remains sound, if we swap the operands of the relation $\mathbf{imp}_m$, we will discuss further theoretical insights.

Theoretical Insights

As discussed in our survey [26], Weyuker was among the first to put learning and testing into relation to each other [292]. She proposed a program-inference-based test-adequacy criterion. In contrast to test-selection criteria, a test-adequacy criterion shall determine whether a passing test suite provides sufficient evidence for the correctness of the SUT. According to Weyuker [292], a test set T should be considered adequate for P if the observations made during the execution of T contained enough information to learn a program P' that is equivalent to P and to the specification that is used for testing. Since checking equivalence is generally undecidable, Weyuker suggests testing-based approximations of that, like for equivalence queries in active automata learning.

More recently, Berg et al. [54] studied the relation between conformance testing and active automata learning, to which they refer as regular inference. Essentially, they conclude that conformance testing and active automata learning perform very similar tasks, with the difference that the former solves a checking problem and the latter solves a synthesis problem. More specifically, they note that it is possible to learn a complete model equivalent to an implementation $\mathcal{I}$ from a complete conformance test suite for $\mathcal{I}$. Likewise, the input sequences executed while learning a complete model of $\mathcal{I}$, form a complete conformance test suite for $\mathcal{I}$. Note that it is in general impossible to derive a complete conformance test suite for black-box implementations, but it is possible if we, for instance, know the maximum number of states; see above for a discussion of the W-method.

While random testing generally does not provide guarantees in terms of (bounded) exhaustiveness, there is a link between random testing and learning in the probably approximately correct (PAC) framework [276]. Angluin already showed that a perfect equivalence oracle can be replaced by random sampling in L^* [37]. Automata learned from sufficiently many samples are approximately correct with some probability $1 - \delta$, whereby the approximation is parameterised with some error-bound ϵ. In fact, efficient exact learning with membership queries and equivalence queries is generally related to PAC learning with only membership queries, see also [212, Theorem 13.3]. Hence, we can give a partial answer to research question **RQ 1.2.** which asks for guarantees provided by randomised testing techniques applied for learning. Randomised testing techniques can provide PAC guarantees.

It should be noted that PAC learnability generally assumes sampling according to an arbitrary but fixed distribution [95]. If we learn a model of some reactive system with random testing in the PAC framework, then we need to keep in mind that this model is only probably approximately correct with respect to the trace distribution used for testing. Thus, such guarantees may provide some false sense of security, if the interaction with a system in practice does not resemble random testing.

Practical Insights

As noted above, test suites are generally not exhaustive, as we cannot test all input sequences. As a result, learned models may be incorrect. This could, for instance, happen if we estimate the maximum number of states m incorrectly, when testing with the W-method [83, 277]. This is problematic, because conservative choices of m, that is, larger than presumably necessary, are not possible due to the exponential growth of test suites in m. Hence, deterministic conformance testing approaches face the issue that they do not scale.

The problem of learning automata from a limited amount of membership queries (test cases) has been addressed in the ZULU challenge [88]. Active automata learning algorithms in this challenge needed to implement equivalence queries through a limited amount of tests. An important insight gained from this challenge has been formulated by Howar et al. [144]. They noted that this challenge required participants to “find counterexamples fast” rather than to “prove equivalence”, for instance, up to some bound with the W-method [83, 277]. For the ZULU challenge, Howar et al. [144] proposed a test-case generation heuristic which, like our mutation-based approach [15, 17], takes inspiration from Rivest and Schapire’s reduced observation tables and the corresponding counterexample processing [240]. Moreover, their generated test cases are of similar shape as test cases created by the random W-method. Each test case has a prefix covering a transition in the hypothesis and a suffix that is chosen according to some heuristic [144]. However, the suffix-selection heuristic does not consider characterising sequences like the random W-method.

More recently, randomised conformance testing has been demonstrated to be successful in practical applications, such as learning a large model of a printer controller [254] or for learning models of TCP clients and servers [112]. The testing technique applied in [112, 254] is similar to a combination of the deterministic W-method and the random W-method, but it applies adaptive distinguishing sequences [183] as suffix if possible.

2.5 Alphabet Abstraction

Active automata learning is usually only feasible for relatively small alphabets and small target automata. This is the case, because the number of tests required for learning grows quickly in the alphabet size k and the target automaton size n. The most efficient active automata learning algorithms, TTT [151] and the RS algorithm [240], require $O(kn^2 + n \log m)$ membership queries in the worst case, where m is the length of the longest counterexample returned by equivalence queries. Depending on the applied testing technique, conformance testing often requires even more tests to be executed and dominates the learning runtime. Grinchtein et al. [54], for instance, consider conformance testing "the true bottleneck of automata learning".

For these reasons, abstraction of inputs and outputs is an essential part of most applications of active automata learning. Let us consider learning-based testing of communication protocols, a popular application domain, to see why abstraction is necessary. The number of valid, concrete messages in protocols such as TCP is virtually unbounded [235]. Even if we ignore message payload, we need to abstract away other parts of messages, like concrete sequence numbers. Generally, we determine a small set of representative messages for abstraction which form the abstract input alphabet. To communicate with the SUL we translate those into concrete messages. Outputs received from the SUL are abstracted as well by discarding message details. Learning from abstract alphabets results in abstract automata with a low number of states.

Mappers. Abstraction and the inverse operation concretisation are often implemented by a mapper component, as described by Aarts et al. [1]. Basically, a mapper acts as a mediator between learner and SUL, as shown in Figure 2.6. It maps abstract inputs to concrete inputs and concrete outputs to abstract outputs during learning. Often, the mapper is considered part of the teacher, thus it would be located inside the teacher box in Figure 2.3.

The following steps implement the execution of a single abstract input i:

1. The learner decides to execute abstract input i and sends i to the mapper.
2. The mapper updates its state depending on i and maps i to a concrete input x according to conditions specified by i.
3. The mapper executes x on the SUL which produces a concrete output y.
4. The mapper updates its state depending on y and abstracts away details of y, mapping it to an abstract output o.
5. The mapper sends o to the learner.
6. The learner records o as abstract output produced in response to the abstract input i.

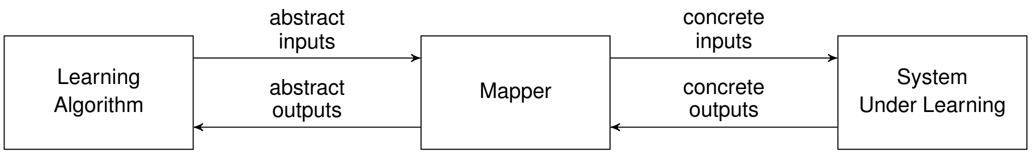


Figure 2.6: Alphabet abstraction through a mapper

Automation and Abstraction in this Thesis. There are approaches to semi-automatically create mappers through abstraction refinement of alphabets [3, 7]. Howar et al. [145] also described alphabet abstraction refinement while integrating the control over abstraction into the learning process. A drawback of these approaches is that they are somewhat restricted with respect to the types of conditions available to express abstraction and concretisation.

In this thesis, we do not employ such techniques, as we focus on other aspects of automata learning, such as timing and uncertainties. Nevertheless, we generally consider abstract automata learning which has implications on the expected size of learned models. Abstract automata learned from deterministic systems usually have less than 100 states [97, 112, 113]. Commonly, abstract automata sizes range from 10 states to approximately 50 states. In experiments presented in the remainder of this thesis, we either create mappers manually or simulate abstract Mealy machines for evaluation purposes. In other words, some evaluation experiments use known abstract Mealy machines as implementations, however, we generally treat these Mealy machines as black boxes during learning.

3 Learning-Based Testing of MQTT Brokers

Declaration of Sources

This chapter discusses our work on learning-based testing of MQTT brokers. It is mainly based on our conference paper presented at ICST 2017 [263], however, it provides some additional content and it contains some improvements of the text. Hence, this chapter covers our practical exploratory research.

3.1 Learning-Based Testing of Network Protocols

In Section 2.4.3, we discussed that conformance testing and learning are related to each other. Berg et al. [54] pointed out that the goal in both areas is to gain knowledge about the behaviour of a black-box system by executing test cases and analysing corresponding observations. In learning, we are interested in the synthesis aspect, that is, we want to learn a model, whereas in conformance testing, we perform a checking task, which means checking conformance to a given model.

This opens up the possibility to combine these approaches. Aarts et al. [4, 5] have for instance shown how to combine learning, testing and verification. They learned the model of a reference implementation of the bounded retransmission protocol and checked equivalence between this model and several mutated (faulty) implementations via two different techniques.

1. The authors performed model-based testing of the mutants using the learned reference model.
2. Additionally, they also learned models of the mutants to check equivalence between the learned specification model and each of the learned mutant models.

In this chapter, we will follow an approach similar to the latter. Our work differs from the work by Aarts et al. [4, 5] mainly in the kind of implementations considered. The authors actually generated Java applications from models with a known structure. Furthermore, the faulty implementations have been created artificially by seeding known errors into the reference model. We, on the contrary, do not know anything about the structure of the analysed implementations and we merely know that they implement a common specification given in natural language.

In order to detect faults in the considered implementations, we thus propose the following learning-based approach. In a first phase, we learn Mealy-machine models of several different implementations of

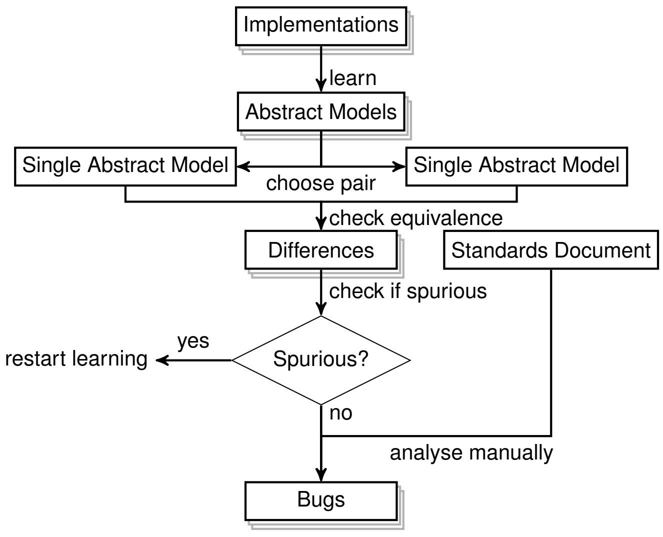


Figure 3.1: Overview of the learning-based testing process

some standardised protocol or operation. These models are then pair-wise cross-checked with respect to equivalence. All counterexamples to equivalence are checked for spuriousness in the next step. We check that, by testing each counterexample on the corresponding implementations. With that, we determine if the counterexample can actually be observed or if it has resulted from incorrect learning. If we have learned incorrectly, we restart learning with more thorough conformance testing to decrease the likelihood of incorrect learning results. After that, we analyse all non-spurious counterexamples manually by consulting a given standards document. This may either reveal a bug in one or both implementations corresponding to the counterexample, or it may reveal an underspecification of some aspect of the standard. The process we follow is also depicted in Figure 3.1.

This approach apparently cannot detect all possible faults since specific faults may be implemented by all examined implementations. In addition to that, the fault-detection capabilities are limited by the level of abstraction used for learning. This gave rise to the research questions **RQ E.1** and **RQ E.2** for our exploratory research that we discussed in Section 1.6.3. **RQ E.1** is whether learning-based testing as discussed above is effective in networked environments. In other words, we want to examine, if it is possible to effectively detect non-trivial faults using our approach despite the necessary severe abstraction. **RQ E.2** addresses limitations of state-of-the-art approaches, that is, we want to investigate what limitations we encounter using the approach outlined above. With respect to the second question, our goal was to identify additional research questions that were briefly discussed in Section 1.6.3 and to identify potential ways to mitigate encountered limitations.

Since we focus on networked environments and in particular the IoT as application domain, we have chosen the MQTT protocol as target for a case study in learning-based testing, as it is used in the IoT. In this chapter, we will analyse the behaviour of brokers implementing MQTT version 3.1.1 [73], a protocol standardised by the International Organization for Standardization (ISO) [153]. Since the MQTT protocol is a lightweight publish/subscribe protocol, it is well-suited for resource-constrained environments such as the IoT. We consider broker implementations, because they constitute central communication units, hence it is essential to assure their correct operation for a reliable communication in the IoT. Basically, brokers allow clients to subscribe and publish to topics. If a message is published, the broker forwards it to all clients with matching subscriptions.

Contributions. The main contribution discussed in this chapter is the presentation and empirical evaluation of the aforementioned approach based on learning experiments with five different black-box systems. More concretely, we learned models of five MQTT brokers and tested these IoT communication systems. In our analysis, we will discuss failure detection with a focus on required effort, issues related

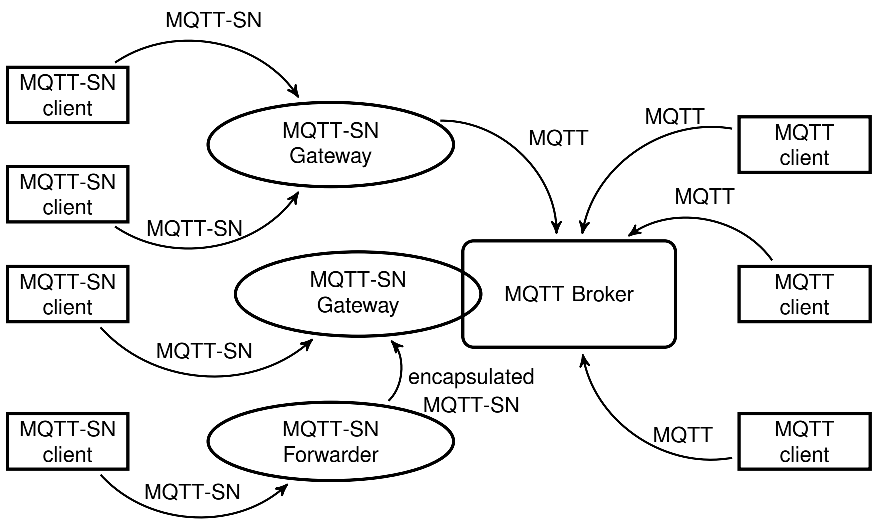


Figure 3.2: An example of the structure of an MQTT network. This example is adapted from Figure 1 of the MQTT-SN specification [257].

to runtime, and general challenges that we faced. Our case study was among the first to apply conformance checking between learned models in a purely black-box setting. Furthermore, it was also one of the first case studies focusing on the verification of implementations of the IoT protocol MQTT.

Chapter Structure. The rest of this chapter is structured as follows. In Section 3.2, we briefly discuss the MQTT protocol. Section 3.3 introduces the approach we follow in our case study in more detail. We will present implementation details and results obtained from learning experiments in Section 3.4. Section 3.5 briefly summarises our work on learning-based testing of MQTT brokers. In Section 3.6, we subsequently discuss the results and findings gained from the performed case study and we also discuss the identified limitations providing the basis for additional research questions.

3.2 The MQTT Protocol

In this section, we will give a short introduction into the MQTT protocol. Unless otherwise noted, we consider version 3.1.1 of the protocol with which we carried out our experiments. For an in-depth description, we refer to the OASIS standards document [73].

The MQTT protocol is a light-weight publish/subscribe-protocol initially developed at IBM and Arcom in 1999 [213]. In March 2013 standardisation at OASIS started and it is now an open standard. It is usually used in machine-to-machine communication. The light-weight nature of the protocol makes it well-suited for resource-constrained environments, such as the IoT. Application areas with particularly strong limitations on resource consumption are considered within a variant of the protocol designed specifically for sensor networks, the MQTT-SN protocol [257].

Generally, MQTT distinguishes between clients and brokers. Additionally, nodes called gateways and forwarders may be present in the network if MQTT-SN is used. Clients can be light-weight devices, such as sensors in the IoT, whereas brokers have to serve multiple clients and also maintain a substantial amount of state related to clients. An example of an MQTT-network structure is shown in Figure 3.2. We see that both, ordinary clients and sensors using MQTT-SN connect to the same broker. As already noted, their role as central communication units makes brokers an ideal subject for our case study. It is of utmost importance to thoroughly test and verify their behaviour in order to ensure reliable communication in IoT infrastructures relying on MQTT.

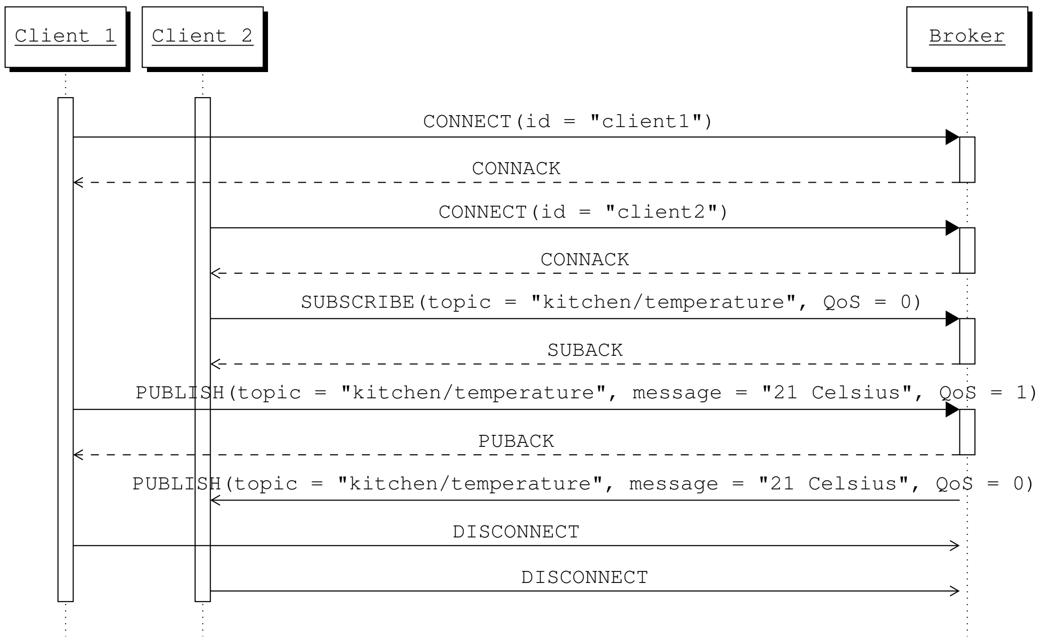

Client 1
Client 2
Broker
CONNECT(id = "client1")
CONNACK
CONNECT(id = "client2")
CONNACK
SUBSCRIBE(topic = "kitchen/temperature", QoS = 0)
SUBACK
PUBLISH(topic = "kitchen/temperature", message = "21 Celsius", QoS = 1)
PUBACK
PUBLISH(topic = "kitchen/temperature", message = "21 Celsius", QoS = 0)
DISCONNECT
DISCONNECT

Figure 3.3: An example of an MQTT-based communication involving two clients and a broker. Some message parameters are abstracted away.

In addition to that, other properties of the MQTT protocol motivated us to select MQTT for our case study. MQTT and MQTT-SN are, for instance, relatively similar. As a result, gateways perform simple tasks so that we can concentrate on testing broker implementations. Another advantage is that there exist several free and open-source implementations of brokers and client libraries.

Basic communication via MQTT is relatively simple. Clients can subscribe to *topics* and publish messages to topics. A broker forwards messages for some topic to all clients with matching subscriptions. An example communication between a broker and two clients is shown in Figure 3.3. In addition to basic publish-subscribe communication, it illustrates the concept of quality of service (QoS) levels. MQTT defines three QoS levels with varying guarantees on message transmission. In this chapter, we consider only the levels 0 and 1. The former ensures that messages are transmitted at most once, whereas the latter ensures that messages are transmitted at least once.

In Figure 3.3, we see the exchange of so-called control packets. The first packets are sent to connect the two clients to the broker. More concretely, the clients send `CONNECT` packets and receive acknowledgements in the form of `CONNACK` packets in return. After `client2` subscribes to the topic `kitchen/temperature` with QoS level 0, the other client, `client1`, publishes the message `21 Celsius` to this topic with QoS level 1. As a result, the publishing client receives an acknowledgement in the form of a `PUBACK` packet. Furthermore, the message is forwarded to `client2`, but with QoS level 0, as that client subscribed with this QoS level. Afterwards, both clients disconnect.

In addition to these simple packets enabling basic publish-subscribe-communication, MQTT also defines some more complex functionality. An example for such functionality are *will* messages. Upon connection, a client may specify a *will* message along with a *will* topic. A broker conforming to the standard must, for instance, publish the will message to the will topic once the connection to the corresponding client is closed due to a protocol error. We will discuss further selected functionalities related to identified faults in Section 3.4. As indicated in Figure 3.3, topics can be hierarchically structured with the delimiter `/`. This is, for instance, used in `kitchen/temperature`. MQTT allows to subscribe to topics using patterns including wild-cards, but we will ignore this functionality, as it would require learning approaches that can cope with data, as presented by Aarts et al. [6] and Cassel et al. [76]. In contrast, we focus on learning of Mealy machines to discover state-machine faults.

3.3 Approach

In this section, we will discuss our approach to learning-based testing of MQTT brokers in more detail. We will start with a discussion on automata learning with a focus on practical considerations to enable learning models of MQTT brokers. These consideration more generally also apply to model learning for reactive systems accessible via network communication.

3.3.1 Learning Environment

Mealy machines are well-suited to model reactive systems, but MQTT-broker implementations may not behave exactly like Mealy machines. This results from the fact that (1) the actual duration of delays between messages may be of relevance, but is usually abstracted away and (2) some implementations do not behave deterministically. Both issues are actually related, as varying network latency may cause the arrival of messages to appear non-deterministically. Additionally, delays may also vary depending on incompletely known influences.

However, MQTT-brokers behave largely deterministic under certain restrictions. For this reason and because we want to explore the limitations of deterministic automata learning, we have chosen Mealy machines as modelling formalism in this case study. Moreover, this enabled us to apply the Java library LearnLib [152] which provides efficient and mature implementations of learning algorithms for deterministic Mealy machines.

Learning Architecture

Two aspects influence the architecture of a learning environment for MQTT. First, we need to account for dependencies between clients. Unlike in pure client/server-settings, such as the TLS protocol [97] or the TCP [7, 112], it is not sufficient to simulate one client to adequately learn a model of the server/broker in MQTT. We need to control multiple clients and record the messages received by each client.

Secondly, we need to cope with the enormous amount of possible inputs, i.e. the large number of packets that we can send to the broker. This, however, is a general problem of model-based testing and active automata learning in the MAT framework, as discussed in Section 2.5. This is an issue for testing and learning of almost all non-trivial systems. We need a way to select representative inputs. To deal with this problem, we introduce a mapper component performing abstraction and concretisation [3, 7, 145]; see also Section 2.5. Comparing automata learning and testing, the abstraction used in learning needs to ensure deterministic observations while some testing approaches do not have this requirement. For instance, property-based testing selects inputs at random from a potentially large set of inputs [84].

As briefly mentioned in Section 2.5, we do not apply approaches to automatically refine abstractions during learning in an iterative manner [3, 7, 145]. We rather use static abstractions, that is, we do not change the mapper throughout the learning phase. If we detected non-determinism arising from abstraction or from the processing of outputs in general, we manually adapted the mapper in an appropriate way to ensure deterministic observations. There may be cases where non-determinism is not caused by abstraction, though, which will be discussed later.

Figure 3.4 shows the architecture of our learning setup. The SUL, an MQTT broker, is shown on the left-hand side. In order to learn its behaviour, we control its environment made up of several clients, each consisting of an adapter block and a client-interface block. The adapters handle low-level tasks, such as serialisation and opening sockets, whereas the client-interface components implement a client library with a simple interface and default values for control-packet parameters.

The right-hand side of Figure 3.4 shows the components responsible for learning. LearnLib [152] implements several algorithms for actively learning Mealy machines. During the execution of such an algorithm, LearnLib interacts with a mapper component by choosing and executing abstract inputs. The mapper concretises abstract inputs by choosing one of the clients and executing some action with

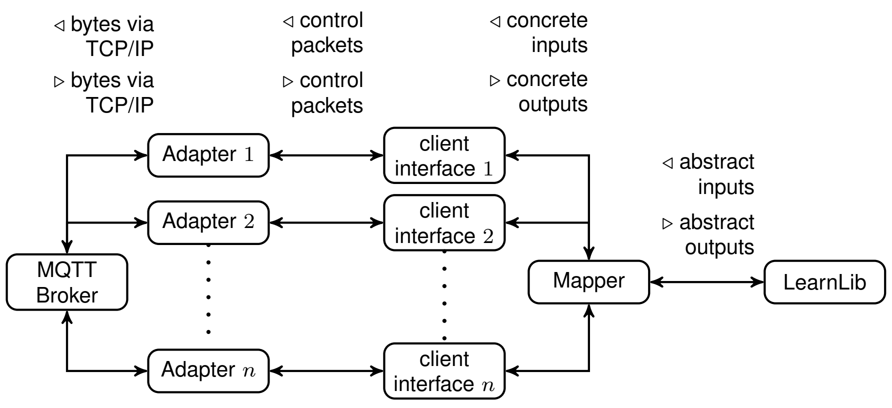


Figure 3.4: The learning environment for learning of Mealy-machine models of MQTT brokers

concrete values. Note that the sending of packets and serialisation tasks performed during an input action are handled by the clients. As soon as outputs are available, the mapper collects them from all clients and abstracts them, creating one abstract output symbol in response to each abstract input.

Practical Considerations

We will now briefly discuss considerations influencing the implementation of the adapter and the mapper component. Both introduce some imprecision into our observations, which is, however, necessary to avoid non-deterministic observations, i.e. cases where two queries sharing some common prefix of inputs produce different outputs.

Timeouts. We already noted that there are delays between the transmission of messages to the broker and the receipt of corresponding responses. These are inevitably present since network communication is involved, which is asynchronous. In addition to that a system may not send any message. It may not produce any output at all. A more faithful model of such a system might therefore, for instance, be a timed automaton [32].

However, instead of changing the modelling formalism we followed a more pragmatic approach. We basically set a configurable timeout on the receipt of messages, as de Ruiter and Poll [97]. All messages received before reaching this timeout are processed by the mapper component to form an abstract output symbol. If some client does not receive any message, we say that the system is quiescent for that client and the mapper produces the corresponding output symbol *Empty*. This is similar to the way the absence of outputs is handled in input-output conformance testing which represents quiescent behaviour by the symbol δ [270]. In Example 2.1, we similarly used the output symbol Q to denote quiescence. Here, we use *Empty*, since there may be an *Empty* symbol for each individual client.

Processing Outputs. We process messages received from brokers in several steps. Following deserialisation in the adapters, the mapper extracts relevant information from messages, such as packet types, abstracting away from details such as identifiers. Each individual client may receive multiple messages in response to a single input sent by one of the clients. Therefore, we process message groups corresponding to each client individually to create one output per client. For that, we sort the messages received by each client alphabetically to ensure determinism. This is necessary as messages may arrive in any order. Afterwards, we concatenate the sorted messages. In other words, we interpret received message sequences as multisets of messages. If a client does not receive any messages, we interpret this as having received a single message *Empty*. The outputs of individual clients are concatenated to create a single abstract output.

While some messages may arrive in any order, the MQTT specification also places restrictions on message ordering [73]. As a result, we lose relevant information through sorting and we might miss some specification violations. To avoid that, we would need to learn non-deterministic models which is potentially more expensive in terms of runtime. Hence, this way of handling outputs represents a tradeoff between completeness and efficiency.

Example 3.1 (Creating Abstract Outputs). Consider a learning experiment in which we control two clients `c1` and `c2`. Now suppose that during a single output query both clients connect to the broker and that both clients subscribe to the topic `top_a`. At this state, `c1` publishes message `mes_a` with QoS level 1 to the topic `top_a`. Consequently, both clients receive `PUBLISH` packets from the broker with message `mes_a`, as both are subscribed to `top_a`. Additionally, `c1` receives an acknowledgement, that is, `c1` receives a `PUBACK` packet.

For learning, we usually extract the packet type from every packet and we extract the topic and the message from `PUBLISH` packets. Hence, for `c1` we would extract relevant information from the received packets to create abstract messages `PUBLISH(top_a,mes_a)` and `PUBACK`, while `c2` receives only `PUBLISH(top_a,mes_a)`. Through sorting the abstract messages received by `c1`, we create the abstract output `PUBACK_PUBLISH(top_a,mes_a)` for `c1`. Combining the abstract outputs from each client through concatenation with delimiter `-`, we create the complete abstract output `PUBACK_PUBLISH(top_a,mes_a)-PUBLISH(top_a,mes_a)`. This output is forwarded to the learning algorithm.

Restrictions Placed on MQTT Functionality. We had to exclude some features of MQTT from our analysis, as they cannot be adequately modelled using Mealy machines. The MQTT specification, for example, includes a ping functionality. To cover this functionality, we would need to learn time-dependent behaviour.

Moreover, there are dependencies between sent and received data. For instance, identifiers sent in acknowledgements should match identifiers of the acknowledged messages [73]. For this reason, we send acknowledgements automatically from the clients to the broker and do not learn behaviour related to that. Such dependencies between different actions could be captured in extended finite-state machines (EFSMs). EFSMs are Mealy machines with parameterised actions, variables storing action parameters, and transitions restricted by guards over state variables and action parameters. The learning tool Tomte [3, 6] is able to learn a restricted class of EFSMs which are expressive enough to model the acknowledgement mechanism. However, we explicitly focus on learning of Mealy machines in this case study to determine how effective this form of learning is.

It should be noted that the excluded features do not affect the parts of MQTT that we learn.

3.3.2 Learning-Based Testing via Cross-Checking Equivalence

We describe our approach to learning-based testing in the following. Roughly speaking, we test for conformance between implementations and flag traces leading to conformance violations as suspicious. This is accomplished by learning models of the concerned systems and subsequent equivalence checking between those models.

For a more detailed discussion on the topic, assume that a model learned with the setup described above faithfully represents the SUL under the abstraction applied by the mapper. As a result, we can execute tests on the model and thereby implicitly perform tests on the SUL under the same abstraction. In other words, we can simulate a testing environment similar to Figure 3.4, but with LearnLib replaced by a conformance testing component.

This differs from the usual approach to model-based testing [274]. In general, a model is assumed to be given which can be used for generating tests and as a test oracle. Generating tests is still possible with learned models, that is, we can generate tests according to some criterion. Using learned models as oracle, however, is problematic, because we cannot be sure whether the model is correct. A model learned by observing a faulty implementation will also be incorrect. To circumvent this problem, we use the learned model of another implementation as oracle.

We thus test whether some learned model conforms to another learned model and thereby we test for conformance between the implementations. Since we do not know anything about the correctness of either implementation, we do not consider conformance violations to be errors, but rather flag traces leading to conformance violations as suspicious. After completing a conformance check, we manually investigate all traces which show non-conformance in order to determine whether any of the implementations violates the specification (in our case the MQTT specification [73]).

This may reveal zero, one or two errors depending on whether the specification allows some freedom of implementation for the corresponding functionality, and whether one or both implementations implemented the functionality in a wrong way. Not all errors are detected for two reasons: (1) it may not be possible to detect some errors because of abstraction and (2) we do not detect an error if it is equally present in all implementations. The first problem is inherent to learning-based testing and to model-based testing in general. The second problem is directly related to our approach. To overcome this issue and to decrease the probability of missing errors we compared the behaviour of five implementations instead of just two.

Input Abstraction. We applied abstraction in our experiments on both inputs and outputs. Example 3.1 demonstrates the output abstraction, in which we simply extract relevant information from packets sent by brokers. In contrast to this, we created abstract inputs by selecting representative concrete inputs to be sent to the broker and assigning an abstract input symbol to each such concrete input. For instance, an abstract input `ConnectC1` may translate to connecting a client with identifier `C1` to the broker by sending the corresponding `CONNECT` control packet. Thus, we fixed the parameters of concrete messages that are sent to the broker. Still, we identified a large number of abstract inputs that are relevant to the MQTT protocol.

Since learning with a large input alphabet containing all identified abstract inputs would not be feasible, we created several smaller input alphabets, each containing only a few abstract inputs. Given these small input alphabets, we learned models using only inputs from a single alphabet. To enable that, we had to implement an individual mapper for each of the alphabets.

We have created the input alphabets in a way such that inputs within some alphabet have relevant dependencies among each other. The effectiveness of fault detection may be impeded by this approach due to implicitly placed assumptions on independence relationships between inputs from different alphabets. Issues such as the effectiveness of the overall approach will be discussed in Section 3.4. A similar approach has been followed by Smeenk et al. [254] for equivalence testing. In addition to a complete alphabet containing all inputs, they identified a subset of inputs relevant to initialisation which they tested more thoroughly.

Testing Process. This form of input abstraction combined with active automata learning and equivalence checking led to the following model-based testing process.

```
For each input alphabet:
  1. Learn a model $\mathcal{M}_i$ of each implementation $i$
  2. For each pair $(\mathcal{M}_i, \mathcal{M}_j)$ of learned models:
    1.1. Check equivalence: $\mathcal{M}_i \equiv \mathcal{M}_j$
    1.2. For each counterexample $c$ to equivalence
      1.2.1. Test $c$ on implementations $i$ and $j$
      1.2.2. If outputs of $i$ and $j$ are not different: $c$ is spurious $\rightarrow$ restart learning
      1.2.3. Analyse manually if outputs of $i$ and $j$ are correct
```

The steps performed for each individual alphabet are also shown in Figure 3.1. Note that if we find a counterexample c to equivalence, we test it on the corresponding implementations. With that, we check if c actually shows a difference between the implementations or if it is spurious. A spurious counterexample may result from an error introduced by learning. This may happen because we approximate equivalence queries by conformance testing. In cases, where we found spurious counterexamples, we restarted learning with more thorough conformance testing. Alternatively, it would be possible to use such counterexamples as counterexamples to equivalence queries in the active-automata-learning process, but we did not implement this direct feedback loop to learning.

As we check for conformance on model level, it is possible to use techniques other than testing. We could use external tools such as CADP to check equivalence [12], encode the problem as reachability problem and use satisfiability modulo theories (SMT) solvers for the task [23], or use a graph-based approach [14, 68]. We actually used a graph-based approach, where we roughly interpreted Mealy machines as labelled transition systems (LTSs) and built a synchronous product with respect to observation equivalence that we use as conformance relation [108, 290]. This will be described in more detail in Section 3.4.

Comparison to Traditional Model-Based Testing. Now that we introduced the approach, we want to discuss the effort required to perform learning-based testing in comparison to the effort required for traditional model-based testing. First, it should be noted that some kind of adapter and abstraction component has to be implemented for both techniques. For this reason, we focus on the effort related to the interpretation of requirements.

Usually in model-based testing, a large set of requirements stated in natural language has to be formalised by creating a system model. This is both labour-intensive and error-prone. In contrast, our learning-based approach does not require such a rigorous formalisation of requirements. To perform the case study, we skimmed through the MQTT specification to get a rough understanding of the protocol in general. Afterwards, we identified interesting interactions between control packets and specific parameters thereof to implement mappers. This can be compared to the definition of scenarios encoding test purposes that are, for instance, used by Spec Explorer [124]. In addition to that, we only had to analyse parts of the specification that correspond to suspicious traces in greater depth. Hence, less manual effort is required.

However, less manual effort comes at the cost of decreased control of the testing process. While it is usually possible to direct the test-case generation through test selection criteria [274], the test cases performed for learning are selected by learning algorithms. This has disadvantages but may also be an advantage. We cannot specifically target certain message flows, but targeting only specific behaviour may also introduce a bias that potentially weakens fault-detection.

The Role of Learning. A question that comes to mind concerns the role of learning in our approach. Why do we actually learn Mealy machines? It would also be possible to generate some test suite, for instance, randomly, execute the test suite on all implementations and check whether differences in outputs exist. However, learning offers two benefits. First, it provides us with a model which can be used in further verification steps, such as targeted model checking of properties derived from requirements [112]. Second, active automata learning essentially defines a stopping criterion. Once we cannot find a counterexample to equivalence between learned model and SUL, we stop testing. Put differently, we can assume to have tested adequately and that we can stop testing as soon as we can derive a system model that is probably correct. Meinke and Sindhu follow a similar approach [206]. They stop testing when learning converges, but they also use other stopping criteria if learning does not converge, such as a bound on the maximum testing time.

3.4 Case Study

In the section, we will discuss our learning-based testing case study. First, we discuss some application-specific details concerning model learning and equivalence checking of learned models. Then, we present the case study results. In this context, we will consider difficulties and issues we faced as well as the manual effort required to classify counterexamples as failures.

3.4.1 Implementation of Learning Environment

The learning part was implemented in Scala using the Java-library LearnLib for active automata learning [152]. Most of the learning-related functionality was thus already implemented and we only had to implement application-specific components such as mappers, and a component responsible for the configuration of learning experiments.

Application-Specific Components

We had to implement the three components shown in the middle of Figure 3.4: the adapter, the client interface, and the mapper. While adapter and client interface merely perform parsing, serialisation, and sending of packets, mappers define a learning target by specifying an input alphabet. Note that we did not use existing client libraries for MQTT, as such libraries may place restrictions on executable input sequences.

As noted in Section 3.3, we used several different abstract input alphabets with interesting dependencies within these alphabets as a basis for learning and consequently testing. For this purpose, we implemented seven different mappers, which all use the same client interface. Because of that, the mappers can be used interchangeably to test different aspects. We will now briefly discuss each of the mappers, with respect to the parts of MQTT they consider. All mappers except for the second allow TCP connections to be reestablished. This means that clients can generally initiate a new TCP connection after they get disconnected from the broker during a query. Put differently, all mappers except for the second allow clients to reconnect multiple times to the MQTT brokers. The mappers developed for our case study are:

Simple. The mapper *Simple* controls one client and offers seven inputs exercising only basic functionality, such as the simplest forms of subscribing and publishing with QoS level 0.

Complex Single Client. This mapper also controls one client, but offers further inputs for publishing and deleting *retained* messages as well as publishing with QoS level 1. It also offers an input for subscribing to a topic with a retained message. A retained message is a message stored by the broker for some specific topic. Each time a client subscribes to such a topic it receives the retained message immediately.

Two Clients with Will. This mapper controls two clients, one of which sets a *will* message while connecting. The same client may close the TCP connection without properly disconnecting, prompting the will message to be published. The other client may subscribe to the topic to which the will message is published.

Two Clients with Retained Will. This mapper is defined similarly to the former, but publishes will messages as retained messages and allows for the deletion of retained messages.

Non-Clean. This mapper controls two clients, one of which connects with a clean session and another one which connects by possibly resuming an MQTT session. An MQTT session is resumed with the *Clean Session* flag set to 0. Learning-based testing with this mapper is supposed to check whether subscriptions of clients are correctly persisted after session termination. A client reconnecting to the broker should be subscribed to the same topics as before it disconnected from the broker.

Two Clients with Same Client Identifier. This mapper also controls two clients, but with the same client identifier. One client starts a clean session upon connection establishment and the other one tries to resume an existing session. With that, we aim to check whether it is possible to take over sessions from a different TCP connection. Additionally, we may learn how brokers implement the termination of clean sessions using this mapper (*Clean Session* flag set to 1).

Invalid. This mapper controls only one client and allows for some invalid inputs to be transmitted, such as subscribing with QoS level -1, publishing to a topic containing a wild card and connecting with an empty client identifier, but with the *Clean Session* flag set to 0. Learning-based testing with this mapper can be considered as robustness testing [150] or fuzzing, because we stimulate brokers with invalid inputs.

These mappers do not cover all aspects of MQTT. It would be possible to define further mappers to target other functionality, but it was not our goal to completely test MQTT brokers. We rather aimed at showing that the proposed approach is an effective aid at finding errors and we assume that experiments performed with seven different mappers provide sufficient evidence for this purpose.

Resets. As mentioned in Section 2.2, active automata learning requires the ability to reset SULs. Basically, we implemented resets of MQTT brokers through the application of several consecutive inputs. More concretely, we performed the following steps to reset brokers during learning:

1. delete all retained messages
2. for each client:
   2.1. reconnect to the broker with the *Clean Session* flag set to 1
   2.2. unsubscribe from all topics (actually not necessary if *Clean Session* is correctly implemented)
   2.3. disconnect again

Alternatively, brokers could be restarted, but the steps above are generally faster, especially because the number of retained messages and active subscriptions is usually low.

Learning & Conformance Testing Configuration

During the course of this case study, we experimented with different learning algorithms, conformance testing algorithms and MQTT implementations. For that, we needed to implement a configuration component that is able to setup learning. However, we did not perform an in-depth evaluation of active automata learning configurations with respect to efficiency as part of the case study. A thorough comparison of learning algorithms and conformance testing algorithms can be found in Section 4.5.

We used the TTT learning algorithm [151] for all experiments presented in the following as it performed best in our experiments. Furthermore, we used the random-walk equivalence oracle provided by LearnLib to perform equivalence queries that we described in Section 2.4.2. This oracle generates random input sequences with geometrically distributed length and it executes these test cases online. We used the LearnLib implementation that places a bound on the maximum number of test steps (abstract inputs) that are executed for an equivalence query. Although random testing is not well-suited to guarantee coverage of some kind, it is a valid choice in our context. The main reason for this is the lack of scalability of complete conformance testing methods, like the W-method [83, 277] or the Wp-method [117]. As mentioned in Section 2.4.2, the number of test cases created by these methods grows exponentially with the estimated maximum model size. Hence, it would be necessary to use low size estimations (depth value in LearnLib) which limits their capability to find counterexamples.

Another reason why we have chosen random testing is that inaccurately learned models due to missed counterexamples may not even be an issue. First, we may detect that we learned incorrectly in our spuriousness check, that is, the check determining whether two brokers truly behave differently for some input sequence. Second, inaccurately learned models might lead to faults being missed, but the overall approach may be effective nonetheless. Since testing is inherently incomplete, we can never find all possible faults, thus potentially missing some faults may not be a severe limitation.

We used random walks with the following settings:

- probability of resetting the SUL (p_{stop} in Algorithm 2.4): 0.05
- maximum number of steps: 10000
- reset step count after finding a counterexample: $true$

At each step, we reset the SUL and start a new test with a probability of 0.05. For each equivalence query, we execute at most 10000 abstract inputs. If we did not find a counterexample within this time, we consider the learned hypothesis correct. Additionally, there is a configuration parameter specifying a broker-specific timeout on the receipt of messages. This is the time we wait for messages to be received in response to a single input (see also Table 3.1).

3.4.2 Checking Equivalence between Models

In the following, we will describe how we check for conformance between two learned models. As usual, observation equivalence serves as conformance relation. As a result from a conformance check, we either output that the checked models are observation equivalent or we present all found counterexamples to equivalence to the user. Conformance checking is basically implemented through bisimulation checks by interpreting Mealy machines as LTSs.

”On the Fly”-Check. We implemented equivalence checking similarly to the on-the-fly bisimulation check proposed by Fernandez and Mounier [108]. For this purpose, we interpret Mealy machines as LTSs by treating each input-output pair labelling a transition in a Mealy machine as a single transition label in the LTS-interpretation.

Following Fernandez and Mounier [108], we build product graphs with respect to bisimulation from two Mealy machines. These product graphs contain special sink states labelled *fail* denoting conformance violations. In order to find counterexamples to equivalence we hence have to find paths to these *fail*-states. More concretely, product graphs contain states formed by pairs of states of both Mealy machines, the fail-states, and transitions between those states. A transition between ordinary states is added for input-output pairs executable in both Mealy machines. If a transition for some input-output pair is executable in only one of the Mealy machines, but not in the other, we add a transition to a fail-state. We check for observation equivalence with that, because we add a fail-state only if there is some input for which the two Mealy machines produce different outputs.

Since we consider deterministic Mealy machines, this check is simple to implement [108]. We implemented it via an implicit depth-first search for fail-states in the product graph. During this exploration, we collect all traces leading to fail-states and present them as counterexamples to the user. Since counterexamples are the only relevant information, we do not actually create the graph. It suffices to store visited states in order to avoid exploring some state twice.

Note that the straight-forward implementation may miss some bugs. Consider a bug which merely produces wrong outputs, but does not affect state transitions. In this case, the bisimulation check will stop exploring at the wrong output and add the reached state to the visited states. If, however, it is necessary to explore the model beyond this state to find another bug, we may miss this bug. Hence, it is possible to miss *double faults* if both faults are reached by a single trace.

To circumvent this problem, we added the possibility to extend counterexamples further until reaching either another difference or a visited state. Actually, the exploration can be continued until a preset maximum number of differences along a trace has been found. With this extended exploration it is thus possible to find multiple counterexamples in cases where the standard exploration would have found only

Table 3.1: Timeout values for receiving messages

Implementation	Timeout in Milliseconds
Apache ActiveMQ	300
emqtt	25
HBMQTT	100
Mosquitto	100
VerneMQ	300

one. This increases the effort required to analyse counterexamples, but it may pay off if additional bugs are found.

The extended exploration did not uncover further bugs in our experiments, but led to a modification of the learning setup. Two models were incorrectly learned because of insufficient equivalence testing during equivalence query. This issue was detected by cross-checking with models of other implementations with extended exploration. Consequently, we increased the number of steps for random equivalence testing during learning and we restarted learning as shown in Figure 3.1.

3.4.3 Experimental Setup

We learned models of five freely available implementations of MQTT brokers. All of them were in active development at the time we performed the case study and wrote the conference paper that we presented at ICST 2017 [263]. The brokers are:

- Apache ActiveMQ 5.13.3[1]
- emqtt 1.0.2 [2]
- HBMQTT 0.7.1 [3]
- Mosquitto 1.4.9 [4]
- VerneMQ 0.12.5p4 [5]

Since all brokers implement version 3.1.1 of MQTT, it was possible to perform all learning experiments in the same way with only minor adaptations. The adaptations basically amount to specifying application-specific timeouts for receiving packets. Table 3.1 shows the timeout values used for the different implementations. We found these values via experiments and should note that they are neither optimal nor do large timeout values indicate poor performance in general. A broker requiring a large timeout may, for instance, provide excellent scalability to large numbers of connections which we did not test. For an in-depth analysis of MQTT timing behaviour, we refer to an evaluation and performance comparison by my supervisor Bernhard Aichernig and Richard Schumi [13].

All experiments were performed with a Lenovo Thinkpad T450 with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz and running Xubuntu Linux 14.04.

[1]`http://activemq.apache.org/`, accessed on November 4, 2019
[2]`http://emqtt.io/`, accessed on November 4, 2019
[3]`https://github.com/beerfactory/hbmqtt`, accessed on November 4, 2019
[4]`http://mosquitto.org/`, accessed on November 4, 2019
[5]`https://vernemq.com/`, accessed on November 4, 2019

3.4.4 Detected Bugs

In the following, we will discuss our findings with respect to error detection in implementations. Altogether we found 17 bugs in all implementations combined. Every implementation contained at least one bug, except for the Mosquitto broker. The detected bugs are actual violations of the MQTT specification [73]. Additionally, we found two cases of unexpected non-determinism in two implementations. These two cases hindered learning, therefore they are not included in the 17 bugs found. Finally, we found a part of the specification which, strictly speaking, none of the implementations implemented correctly. However, four of the implementations showed behaviour users might expect to see, thus we consider these implementations to be correct. One implementation, however, showed unexpected faulty behaviour. Hence, **we actually found 18 bugs altogether**.

Four of the bugs correspond to issues already reported by other users. The remaining bugs were reported by us to the developers of the brokers and have been reviewed. As a result, the bugs have mostly been fixed. In the following, we will review two examples of bugs we found in more detail. In contrast to our conference paper [263], we also provide a complete list of all bugs, where each one is briefly discussed.

Violations of the Specification

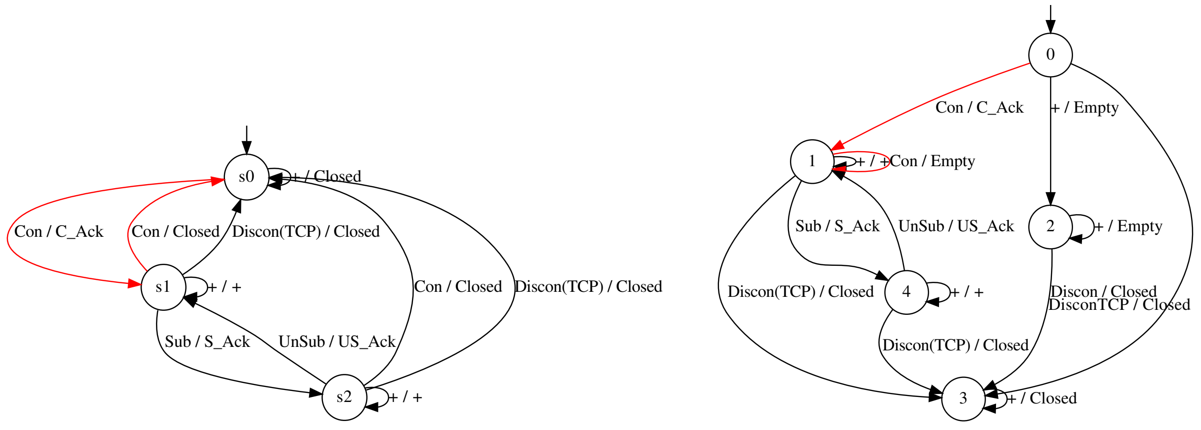
A Simple Error. A simple example of a violation of the protocol specification can be found by considering the behaviour of the HBMQTT broker with respect to the functionality covered by the *Simple* mapper. Figure 3.5 shows the models learned by observing the Mosquitto broker and the HBMQTT broker with abbreviated action labels. A counterexample to equivalence is `CONNECT` · `CONNECT`, which is shown in red in both models. For Mosquitto, we observe the output `C_Ack` · `ConnectionClosed` and for HBMQTT, we observe the output `C_Ack` · *Empty*. HBMQTT acknowledges the first connection attempt and ignores the second one by not producing any output and it actually does not change its state as well.

The MQTT specification states that Mosquitto's behaviour is correct whereas HBMQTT behaves in an incorrect way [73]:

> "A Client can only send the CONNECT Packet once over a Network Connection. The Server MUST process a second CONNECT Packet sent from a Client as a protocol violation and disconnect the Client [MQTT-3.1.0-2]."

An Error Requiring Two Clients. The first example of an error is admittedly rather simple. However, we also found more subtle bugs. A strength with regard to error detection of our approach is that Mealy machines are input enabled. Therefore, we do not only learn and test the usual behaviour, but also exceptional cases. Using the mapper *Two Clients with Retained Will*, we found an interesting sequence which uncovered errors in both, emqtt and ActiveMQ. The sequence is as follows:

1. A client connects with client identifier `Client1`
2. Another client connects with client identifier `Client2`, specifying `bye` as retained will message for topic `c2_will`
3. `Client2` disconnects unexpectedly prompting the broker to publish the will message
4. `Client1` subscribes to `c2_will`
5. `Client1` subscribes again to `c2_will`



(a) Mosquitto model **(b)** HBMQTT model

Figure 3.5: Models of two MQTT brokers learned with the *Simple* mapper, where red transitions highlight a difference corresponding to a bug. Some inputs and outputs have been combined, which is denoted by the '+'-symbol.

The responses to the first two steps are delivered as expected. The tested brokers sent acknowledgements to the clients in response to their connection. They also sent the will message to `Client1` in the fourth step which is correct as well. In the fifth step, Mosquitto behaved differently from emqtt and ActiveMQ. While emqtt and ActiveMQ did not resend `bye` to `Client1`, the Mosquitto broker sent `bye` again.

The behaviour of ActiveMQ and emqtt was incorrect according to [MQTT-3.8.4-3] [73] which states that repeated subscription requests must replace existing subscriptions and that "any existing retained messages matching the Topic Filter MUST be re-sent".

Further Errors. We have just discussed the first three errors, with one error being equally implemented by two different brokers which is why we counted it twice. The complete list of errors is:

- 4th error: ActiveMQ responded to ping requests from clients that have not sent a `CONNECT` packet before. This violates [MQTT-3.1.0-1] and [MQTT-4.8.0-1] of the specification [73].
- 5th & 6th error: emqtt and VerneMQ implemented a detail related to the *Clean Session* incorrectly. The error could be triggered by the sequence: (1) connect with client identifier `c1` and *Clean Session* set to true, (2) subscribe with client identifier `c1` to `top`, (3) connect again with `c1`, but with *Clean Session* set to false, and (4) publish some message to `top`.

  In step (3), the brokers correctly close the first session and create a new one with the same client identifier. According to [MQTT-3.1.2-6] [73], state data from sessions with *Clean Session* set to true must not be reused in subsequent sessions, thus the subscription from the first session must be discarded. However, emqtt and ActiveMQ forward the published message to the new client `c1` in step (4) which is incorrect, as this client has no valid subscription at this point.
- 7th & 8th error: ActiveMQ and emqtt did not resend retained messages if clients subscribed twice to the same topic. In other words, if a client subscribed twice to a topic with a retained message, it should also receive the retained message twice, but ActiveMQ and emqtt sent it only once. This behaviour violates [MQTT-3.8.4-3] of the specification [73].
- 9th error: HBMQTT kept TCP connections open in case of protocol violations, but ignored all subsequent packets sent by clients. This violates [MQTT-3.1.0-1] and [MQTT-4.8.0-1] of the specification [73].

- 10th error: VerneMQ did not publish empty retained messages which is incorrect according to [MQTT-3.3.1-10] [73].
- 11th error: in learning experiments with mapper *Two Clients with Retained Will*, VerneMQ published the retained will message even if the corresponding client disconnected gracefully. This is incorrect according to [MQTT-3.1.2-8] [73].
- 12th error: HBMQTT implemented hierarchical topics incorrectly. We found this error unintentionally, as we actually did not target data-related behaviour. However, we observed that by subscribing to `my_topic`, we would receive messages published to `my_topic/retained` as well. This behaviour is incorrect, as the topic `my_topic` does not match `my_topic/retained` [73]. Note that `my_topic/#` would match `my_topic/retained`, because of the wild-card character `#`.
- 13th error: VerneMQ showed another error related to the 5th error. The error was observed when executing the following sequence: (1) connect with client identifier `c1`, (2) subscribe with `c1` to `top`, (3) connect again with client identifier `c1`, (4) subscribe again with `c1` to `top`, and (5) publish to `top`. The message from step (5) was published twice by VerneMQ. Hence, the broker stored two subscriptions for the same client and the same topic. The second subscription from step (4) should replace the first subscription [73].
- 14th error: ActiveMQ allows publishing to topics containing wildcards. This is not allowed by [MQTT-3.3.2-2] [73]. We found this error by testing with the mapper *Invalid*.
- 15th error: HBMQTT acknowledges subscriptions with QoS level 3 which must be treated as a protocol violation according to [MQTT-3-8.3-4] [73]. The MQTT specification defines only the QoS levels 0, 1 and 2.
- 16th error: HBMQTT ignores `PUBLISH` packets with QoS level 3, but according to [MQTT-3.3.1-4] [73], the network connection must be closed in such a situation.
- 17th error: emqtt accepts connections with empty client identifier and with *Clean Session* set to false. According to [MQTT-3.1.3-8] [73], brokers must acknowledge such connections, but immediately close the network connection.
- 18th error: as noted above, all tested brokers implemented one aspect of the protocol non-standard conform. The clause [MQTT-3.3.1-2] [73] states that the `DUP` flag must be set to false in `PUBLISH` packets with QoS level 0. In general, brokers and clients need to close the network connection, when they observe a protocol violation [73, MQTT 4.8.0-1]. Hence, brokers should close the network connection when [MQTT-3.3.1-2] is violated by a client.

  However, none of the tested brokers did that. ActiveMQ, Mosquitto, VerneMQ, and emqtt ignored the `DUP` flag and published messages with QoS level 0. In contrast to this, HBMQTT ignored such malformed `PUBLISH` packets altogether, without closing the network connection. As the `DUP` actually only provides additional information, we consider HBMQTT's behaviour incorrect and the behaviour of the other four brokers to be not faulty. Strictly speaking, the behaviour of the other brokers is not correct either, but it is more likely to conform to user expectations.

In summary, we found four errors in ActiveMQ, four errors in emqtt, six errors in HBMTT, four errors in VerneMQ, and none in Mosquitto.

Non-Determinism

An issue we faced during our experiments was that Mealy-machine learning algorithms cannot cope with non-deterministic behaviour. In our setup based on LearnLib, LearnLib throws an exception and stops learning as soon as it detects non-deterministic behaviour. Thus, we may waste test time in such cases. The only information we gain from such experiments is that non-determinism affects the experiments accompanied with sequences witnessing non-determinism. These witness sequences are pairs of input/output sequences with the same inputs but with different outputs.

Non-determinism may result from several sources including:

- learning setup
- time-dependent issues
- actual non-determinism displayed by implementations

In the first case, it is actually beneficial that learning stops, as the setup should not introduce non-determinism. It is likely to contain errors in this case. One issue related to timing is the unknown time it takes for a broker to respond. In order to avoid non-deterministic behaviour in this regard, we implemented the aforementioned timeout on the receipt of messages. We thus introduce imprecision to overcome time-related non-determinism. Considering that TCP is actually a reliable transport protocol and that the user datagram protocol (UDP) is often used in the IoT, time-related non-determinism is likely to be a more severe issue in other IoT protocols.

Implementations may also show truly non-deterministic/uncertain behaviour. In this case, the repeated execution of some input sequence under the same conditions may lead to different results. Unfortunately, it is not possible to adapt our learning setup with reasonable effort to account for such cases of non-determinism. For this reason, we could not successfully perform 3 out of 35 learning experiments. We actually evaluated further MQTT implementations in addition to those listed in Section 3.4, which we excluded from the experiments because of non-deterministic behaviour. These additional implementations showed non-deterministic behaviour in learning experiments with the simplest mapper. Considering that, we conclude that learning-based verification would greatly benefit from being able to learn models capturing uncertainties.

Discussion

In the following, we will review our experiences using the proposed approach with special regard to manual effort. In this context, we will also recapitulate some already discussed issues affecting the required effort.

In the initial phase, some experimentation was necessary to understand how to define mappers with reasonable complexity. By this we mean mappers that are not trivial, but which allow for learning to be completed in an acceptable amount of time. This usually does not require a substantial amount of human labour, but requires computation time as experiments have to be executed repeatedly with at least one implementation. Defining mappers can probably best be compared to defining test-case specification scenarios, like for testing with Spec Explorer [124].

We also spent a considerable amount of time analysing suspicious traces, a task not needed in traditional model-based testing as requirements have to be formalised beforehand. In this context, we made the observation that a single bug usually results in several counterexamples to equivalence. In addition to the standard equivalence check, we also used the extended bisimulation check to avoid missing bugs. This led to additional manual effort.

Consider, for instance, the first bug discussed and highlighted in Figure 3.5. Essentially the same bug can be detected by analysing the counterexample *Connect* · *Subscribe* · *Connect*. Wenn cross-checking the models in Figure 3.5, we actually found seven counterexamples with the bisimulation check and we found 24 counterexamples by extended exploration. All these counterexamples point to only two different bugs. The 10th error which causes VerneMQ to not publish empty retained messages exemplifies that the extended check may not cause any overhead. Checking equivalence between a model of Mosquitto and a model of VerneMQ finds four counterexamples with either of the checks.

At the current stage, we implemented a mechanism to manually define filters to hide counterexamples matching a specified pattern. Thus, it is possible to analyse a counterexample, find a bug and specify a pattern to exclude similar counterexamples. Coarse patterns may lead to bugs being undetected, therefore we did not use filters in our experiments.

However, a reduction of counterexamples or some kind of automated partitioning into equivalence classes of counterexamples may be crucial for a successful application of the approach to more complex systems. It would, for instance, be possible to follow an approach similar to *MoreBugs* [147] to implement such a technique. The MoreBugs method tries to infer a bug pattern from a failing test case and avoids testing the same pattern repeatedly.We could group counterexamples by matching them to inferred patterns and present only one counterexample per group to users. Especially the parallel-composition-based pattern-inference of MoreBugs seems promising in our use case. Since Mealy machines are input-enabled, inputs result in self-loops in many states which causes counterexample traces to be interleaved with non-relevant inputs.

We noted in Section 3.3 that it may be possible to learn an incorrect model if the equivalence oracle, which is only an approximation, provides a wrong answer. Therefore, we test counterexamples as stand-alone tests to see whether they are spurious. Note that test cases corresponding to non-spurious counterexamples form a regression test suite focused on previously detected bugs.

A more problematic scenario is that we may learn incompletely and consequently miss erroneous behaviour. However, on the one hand we have seen that bugs usually result in several counterexamples which lowers the probability of missing bugs. On the other hand, testing is inherently incomplete, so there is always the possibility that we do not detect all bugs.

We conclude that it is possible to find non-trivial bugs in protocol implementations with reasonable effort despite necessary harsh abstraction. Testing more complex systems may be complicated by the large number of counterexamples that need to be analysed. Tasks other than that have comparable counterparts in traditional model-based testing. It should be emphasised that the initial effort to set up a learning environment is relatively low due to the flexibility and ease of use of LearnLib [152].

3.4.5 Efficiency

We faced an issue related to runtime during learning. Learning generally took a long time. Table 3.2 and Table 3.3 show runtime measurement results for learning with the mapper *Simple* and the mapper *Two Clients with Retained Will*, respectively.

The results include the number of states in the learned models, the time and number of queries needed for output queries (*OQ time[s]* and *OQ # queries*), and the time and number of queries needed for conformance testing (*CT time[s]* and *CT # queries*). The number of queries for conformance testing is the combined number of test cases executed during all equivalence queries performed throughout learning. The number of equivalence queries represents the number of rounds of every learning experiment. The last row of the tables contains this number.

It can be observed that we actually deal with relatively simple models. The largest models have eighteen states and the larger of the two input alphabets contains nine input symbols. Despite the possibility to learn much larger models with active automata learning, such as Merten et al. who achieved to learn a model with over a million states [208], we faced efficiency issues while learning much smaller models. This can be explained by considering the long runtime of individual tests/queries. In contrast to this, Merten et al. [208] used an extremely fast membership oracle in their setting.

In our setting, tests may take several seconds since we wait up to 600 milliseconds for outputs from two clients in response to a single input (300 milliseconds per client). Thus, we see similar learning performance as when learning the TLS protocol [97]. Unlike in the context of learning TLS, we do not stop testing once a connection is closed, thus testing MQTT is more expensive in terms of test-execution time. This is required to learn behaviour relevant to persistent session state.

Table 3.2: Runtime measurement results obtained by learning with the mapper *Simple* with an alphabet size of 7

	ActiveMQ	emqtt	HBMQTT	Mosquitto	VerneMQ
# states	4	3	5	3	3
OQ time[s]	59.72	3.87	31.94	14.01	43.91
OQ # queries	88	59	110	56	57
CT time[s]	914.18	78.3	491.06	278.21	915.77
CT # queries	525	519	482	487	490
# equivalence queries	4	3	4	3	3

Table 3.3: Runtime measurement results obtained by learning with the mapper *Two Clients with Retained Will* with an alphabet size of 9

	ActiveMQ	emqtt	HBMQTT	Mosquitto	VerneMQ
# states	18	18	17	18	17
OQ time[s]	1855.55	167.32	557.14	641.89	1570.8
OQ # queries	732	735	640	730	625
CT time[s]	4787.92	481.36	2022.47	1612.59	4355.97
CT # queries	672	816	613	670	658
# equivalence queries	13	12	11	9	11

The drastic influence of testing runtime can be seen in experiments performed with ActiveMQ and VerneMQ, as they require the largest timeout on the receipt of broker responses. Even the 3-state VerneMQ model learned with *Simple* mapper takes almost 16 minutes to learn (see Table 3.2). The longest experiment, learning a model of ActiveMQ with the mapper *Two Clients Retained Will*, takes more than 110 minutes and resulted in a model with only eighteen states (see Table 3.3). These high runtimes for learning comparably simple models make apparent that there is a need to keep the number and length of queries to be executed as small as possible. This can, for instance, be achieved via domain-specific optimisations, heuristics and smart test selection [149, 254] or via algorithmic advantages [151].

3.5 Summary

We presented a learning-based approach to semi-automatically detect failures of reactive systems in this chapter. We evaluated the effectiveness of this approach by means of a case study. In total we found 18 faults in four out of five MQTT brokers.

More concretely, we learned abstract models of MQTT brokers. Based on that, we identified observable differences between the considered implementations in an automated manner. Since these differences are likely to show erroneous behaviour we inspected them manually to determine whether they show specification violations.

The performed case study was among the first to apply learning-based testing on reactive systems implemented independently by open-source developers and it was the first attempt at model-based testing MQTT brokers. We showed that the proposed approach can effectively detect bugs without requiring any prior modelling.

3.6 Results and Findings

In the following, we will analyse our case study in learning-based testing with respect to the two research questions defined for the exploratory research phase.

RQ E.1 Is learning-based testing effective in networked environments such as the IoT?

Since we found 18 protocol violations with comparably low effort, we conclude that learning-based testing is effective. While the approach can generally be applied to any type of reactive system for which there exist multiple implementations, it is especially well-suited for networked environments. This is the case, because network protocols can be modelled abstractly with a low number of states, making active automata learning feasible. Moreover, well-defined standards are likely to exist for common network protocols.

The effectiveness of learning-based testing in networked environments is also apparent in other case studies [97, 112, 113, 253]. However, we have specifically chosen the MQTT protocol for its relevance to the IoT and because we had little prior knowledge about the protocol. Our goal was to truly examine how much effort is necessary to apply the approach in an unknown environment. As mentioned in Section 3.4, implementing the learning setup required low effort, as LearnLib [152] is easy to use. In comparison to model-based testing, which usually requires a complete formalisation of requirements, the requirements analysis was significantly less labour-intensive, as it was focused on counterexample traces. Putting the outcome of finding several true errors into relation with low testing effort reinforces the claim that learning-based testing is effective.

RQ E.2 What are the shortcomings of existing automata-learning approaches in practical applications?

In addition to investigating the practicality of learning-based testing in general, we wanted to determine potential limitations with the performed case study in order to identify directions for further research. We concretely formulated these directions for further research as research questions separated into three groups. They are discussed and summarised below and can be found in Section 1.6.4.

Uncertainties. We observed uncertain behaviour. As noted in Section 1.6.3, we use the term uncertain behaviour for both non-deterministic and stochastic behaviour. Such behaviour exists when a single input sequence may produce different outputs when applied multiple times. Not being able to capture uncertainties directly affects effectiveness. We were not able to perform all experiments for some implementations.

Since it is common for complex reactive systems to show uncertain behaviour, we deemed it important to investigate ways to learn such behaviour, especially in test-based settings. There are basically two main approaches to model uncertainties, either through non-determinism or probabilistically in stochastic models. Active non-deterministic learning techniques [161, 283] suffer from the fact that they abstract from the observed frequency of actions. As a result, it may be hard to decide when all possible outcomes have been observed. Alternatively, learning methods for stochastic behaviour may be used [74, 199]. Such methods have been studied in the grammatical-inference community [95] and have been analysed with respect to PAC guarantees [77]. PAC learning provides quantitative correctness guarantees in relation to the number of sampled system traces. While these methods have also been applied in a verification context [199], most of them are passive. This means that they learn from given system traces. This poses the question of how to actively learn models of stochastic behaviour via testing.

In summary, we face three questions which are in short: (1) are non-deterministic or stochastic models more appropriate? (2) When do we have observed sufficient data to create a faithful model? (3) If we model uncertainties stochastically, how can we learn such models via active testing?

Timing-related Behaviour. Certain aspects of the MQTT functionality depend on timing. We had to exclude these aspects from our analysis, because Mealy machines cannot precisely capture timing aspects. The inability to model timing therefore impedes the effectiveness of testing. This issue is likely to be more severe in lower-level protocols and in networks with transport protocols less reliable than TCP, which is used by MQTT.

Existing learning approaches for timed models focus on relatively restricted types of models, such as event-recording automata [125, 126] and real-time automata [281, 282]. General timed automata are more expressive and provide more degrees of freedom, making learning also more difficult. In this context, Verwer et al. [281] provide hardness results for learning models that are more expressive than real-time automata. Grinchtein et al. [126] note that general timed automata do not have canonical forms, which complicates learning.

In general, our goal is to enable learning-based analysis with prior knowledge as little as possible. This means that we want to learn timed models that are as expressive as possible. Given that existing techniques support only restricted types of models, we decided to investigate (1) whether alternative techniques, such as metaheuristics, are applicable for learning expressive timed models. Moreover, we decided to examine (2) what assumptions about the SUL are necessary to enable learning.

Runtime. We discussed that learning often took a relatively long time to complete in Section 3.4.5. This is mainly due to the high test-execution time caused by the timeouts on the receipt of messages from the broker. In Table 3.2 and in Table 3.3 we see that we spent the most time on conformance testing. This matches our expectations, because we applied the TTT algorithm which aims to keep data structures small and consequently output queries short. Hence, it would be worthwhile to develop more efficient conformance-testing techniques.

Recall that we applied random testing. We did that because deterministic testing with the W-method [83, 277] did not complete in reasonable time. For instance, learning ActiveMQ models with complex mappers did not finish in less than a day, unless we set the depth parameter to a value that resulted in learning incorrect models. Instead we performed random walks for testing and we restarted learning with more thorough testing when we found spurious counterexamples. This means that a lower number of random test cases would likely cause incorrect learning.

Given these observations, we identified four research questions. (1) Our observations suggest that random testing can be an appropriate choice, because deterministic testing did not scale, but is this generally true? Suppose that only random testing can be applied, as deterministic testing does not terminate. (2) Can we still give guarantees in that case? In the MQTT case study, we are able to guarantee that a non-spurious counterexample reveals a difference between two implementations, although the learned models may be incorrect. (3) Is it possible, though, to reliably learn correct models with random testing? As mentioned above, we could not have executed a lower number of test cases, without risking to learn incorrectly. This raises the questions whether we can generate test cases in a smarter way, to cover system behaviour thoroughly with a lower number of test cases. (4) Since Bernhard Aichernig's group successfully combined random testing with fault-based testing [22], we decided to investigate if this combination is also effective in active automata learning.

A concluding remark on research priority. All the identified limitations impede the effectiveness of learning-based testing in some way. The inability to capture uncertainties and timing behaviour requires certain systems and some specific aspects of systems to be ignored. High test-execution times also impede effectiveness, by restricting the applicability of learning-based testing to systems with small abstract state space. This issue affects all types of systems. As a consequence, we decided to investigate test-case generation techniques as a first step, to gain insights into testing in a learning-based setting.

4 Efficient Conformance Testing in Active Automata Learning

Declaration of Sources

In this chapter, we discuss our work on efficient conformance testing in active automata learning. This mainly includes our fault-based testing approach for active automata learning as well as an evaluation of various active automata learning configurations with respect to efficiency. We first presented the proposed testing approach at the NFM 2017 [15]. Following the presentation at the symposium, we extended our work on this topic to a journal article, which is included in the special issue of the *Journal of Automated Reasoning* on the NFM 2017 [17]. This chapter is mostly based on the extended article, which has already been published online. Section 4.5 is based on Felix Wallner's bachelor's thesis on benchmarking active automata learning configuration [288].

4.1 Introduction

This chapter addresses efficiency in active automata learning. Since executing equivalence queries usually dominates the runtime of active automata learning, we focus on conformance testing in active automata learning. Our main contribution in this context is an efficient fault-based conformance testing technique for the implementation of equivalence queries. Hence, this technique is presented in this chapter. In addition, we consider efficiency of active automata learning in general and evaluate various learning configurations with respect to efficiency.

In the following, we will briefly discuss previous research on the implementation of equivalence queries, highlighting some of the findings that relate to efficiency. Subsequently, we will present the idea behind our fault-based testing approach. Before going into details, we should comment on terminology: we mainly consider *efficiency* with respect to the required testing budget, thus efficient testing techniques require a low number of test cases/steps to be executed. Related to that, we say that test cases are *effective* if they are likely to detect conformance violations. Hence, the execution of effective test cases leads to efficient testing.

Table 4.1: The number of test cases generated by the W-method and the Wp-method during learning the car alarm system via L^*_M

depth	W-method		Wp-method	
	1st eq. query	2nd eq. query	1st eq. query	2nd eq. query
1	136	252	130	200
2	680	1260	546	840
3	2856	5292	2210	3400

Test-Based Equivalence Query Implementations

Peled et al. [230] introduced *Black Box Checking* as a technique for model checking black-box systems. As one of the first to combine active automata learning and formal verification, the authors proposed to implement equivalence queries via conformance testing. In particular, they suggested to use the W-method [83, 277]. It should be noted that optimisations of the W-method, such as the partial W-method [117] or an approach by Lee and Yannankakis [54, 184] could have been used as well. Similar to the W-method, these techniques achieve completeness relative to some bound m on the maximum number of states of a minimal system model. In comparison to the W-method, these alternative techniques require fewer tests to be executed with the reduction being only polynomial [54]. All three techniques have the same worst-case complexity that is exponential in the bound m. The exponential size of the constructed test suites severely limits the practical applicability of automata learning combined with these testing techniques. Additionally, we usually do not know the bound m, as we take a black-box view of systems. This bound was originally derived as an estimation given a known system design [83], while we, however, strive for learning the system design without prior knowledge. Without such knowledge it is generally impossible to implement perfect equivalence queries [54]. Due to these issues, implementing equivalence queries can be considered “the true bottleneck of automata learning” [54]. The following example shall demonstrate the large cost of deterministic conformance testing in terms of test-suite size.

Example 4.1 (Testing the Car Alarm System). The following example demonstrates testing with the W-method and the partial W-method by applying them to learn the car alarm system shown in Example 2.1. The first hypothesis learned with L^*_M [246] is shown Figure 2.4. This hypothesis has only one state less than the complete true model that has five states. Hence, we need to execute at most two equivalence queries: one producing a counterexample and adding a new state and another confirming that the final hypothesis and the SUL are equivalent.

Table 4.1 shows the number of test cases generated by the W-method and the Wp-method for both equivalence queries with varying depth values. The depth value d is the difference between assumed bound m on the number of states and the number of states of the hypothesis under test; see $d = m - |Q|$ in Line 3 of Algorithm 2.2 which specifies how many infix sequences are generated and tested. Although the Wp-method requires fewer tests than the W-method, we see that the number of test cases is mainly influenced by the depth parameter d. We also observe that the number of test cases is quite large, even for this simple example with only four inputs. If we set d to three, we generate 3,400 test cases just for the second equivalence query with the Wp-method. Note that by setting d to three we assume that the system has at most $m = 8$ states, since the second hypothesis has five states. In contrast to the large number of test cases for equivalence queries, we need to perform only 84 output queries.

Despite the apparent hardness of implementing equivalence queries, active automata learning has been successfully applied in verification and testing [26]. Hence, we shall examine properties of successful applications of active automata learning. In Chapter 3, we have shown that learned models do not need to be complete. In our case study on learning-based testing of the MQTT protocol, it was sufficient for models to capture enough information such that non-trivial faults could be detected. Hence, it is not necessary for equivalence queries to be perfect. However, in order to capture as much information in learned models as possible, equivalence queries should be as complete as possible. Given the limited time for testing in practice, testing approaches should aim at covering as much behaviour as possible.

Learning with Limited Testing Budget

The ZULU challenge [88] addressed the issue of a limited time budget for learning, by limiting the number of test cases that can be executed for learning [144]. More concretely, competitors learned finite automata from a limited number of membership queries without explicit equivalence queries. Equivalence queries thus had to be approximated through clever selection of membership queries. This led to a different view of the problem: rather than “trying to prove equivalence”, the new target was “finding counterexamples fast” [144]. Thus, one of the insights from this challenge was that techniques such as the W-method are not feasible in most scenarios. Howar et al. further note that it is necessary to find counterexamples with few tests for automata learning to be practically applicable [144]. Following this reasoning, they described effective test-case generation heuristics for active automata learning. These heuristics produced random test cases matching counterexample patterns inspired by Rivest and Schapire’s improvements of the L^* algorithm [240] (see also Section 2.3.1).

More recently, Smeenk et al. [254] applied a partly randomised conformance testing technique to actively learn large models of embedded control software. They generate test cases similarly to the random W-method discussed in Section 2.4.2, without using characterising sequences [83, 277]. In order to distinguish states with as few test cases as possible, they applied a technique to determine an adaptive distinguishing sequence described by Lee and Yannakakis [183]. The proposed testing technique actually applies deterministic conformance testing relative to some small bound m and randomised conformance testing relative to a larger bound. The same technique has also been used to learn models of TCP implementations [112] and of SSH implementations [113].

We see that randomised conformance testing has successfully been applied in active automata learning. Although both techniques discussed above are randomised, they actually achieve some form of model coverage. Test cases generated by Howar et al. [144] and by Smeenk et al. [254] start with a prefix leading to a transition/state in the current hypothesis. After sufficiently many test cases have been executed, it is likely that all transitions/states of the current hypothesis have been covered. Howar et al. [144] address coverage explicitly in their heuristics. During the creation of prefixes, they favour prefixes that cover transitions that have not been covered before.

Fault-Based Conformance Testing for Learning

Like the heuristics by Howar et al. [144], the approach to conformance testing presented in this chapter is randomised, coverage-guided, and inspired by Rivest and Schapire’s improvements of L^* [240]. Despite these similarities, we target a different notion of coverage and we implement randomisation quite differently, as we do not use fixed counterexample patterns to generate test cases. We propose an implementation of equivalence queries based on mutation testing [155], more specifically on model-based mutation testing [22]. In this fault-based testing approach, a given model is mutated by injecting known faults into it. This creates a set of mutated models, called mutants. Test-case generation then creates test cases that can distinguish mutants from the original model. By executing those test cases on the SUT, it is possible to either detect faults corresponding to mutants or to demonstrate that these faults have not been implemented.

In the approach presented in this chapter, we do not aim at covering all possible mutants, since this would not scale, much like the W-method.[1] Our goal is rather to cover as many model mutants as possible. More concretely, we combine random testing and mutation analysis to achieve high variability of test cases and to address coverage appropriately. In short, we first create mutants of the current hypothesis that represent potential succeeding hypotheses. Then, we create random test cases and select

[1]Note that this holds for automata learning, where the number of relevant mutants is indeed very large. Covering all mutants is feasible in other scenarios, though.

the ones covering the most mutants. In other words, we select test cases that are likely able to distinguish the current hypothesis from a future hypothesis which is closer to the SUL. Our mutation technique is inspired by the effect of counterexample processing presented by Rivest and Schapire [240]. Extracting and adding a counterexample basically splits states in the current hypothesis. We mimic this operation through mutation.

The combination of model-based mutation testing and random testing was found to be effective in previous work of Bernhard Aichernig, the supervisor of this thesis [21, 22]. Generally, random testing is able to detect a large number of mutants fast, such that only a few subtle mutants need to be checked with directed search techniques. Since we follow the spirit of the ZULU challenge [88, 144], we do not aim at complete coverage of all mutants, as this might require too many test cases. Therefore, we do not perform directed search for test-case generation optimising mutation coverage. Still, the mutation coverage by randomly generated test cases provides a certain level of confidence. An analysis of mutation coverage allows to guarantee that covered mutations do not affect the learned model.

We implemented our testing technique in the LearnLib library [152] to use it in active automata learning. We evaluate it by comparing it to other testing techniques and show that the cost of testing during learning can be significantly reduced while still learning correctly. In particular, we compare our approach to the partial W-method [117] and to the testing technique applied by Smeenk et al. [254], but we apply this technique fully randomised. In our evaluation, we empirically determine the lowest number of test cases that are required to reliably find counterexamples for all incorrect hypotheses. The evaluation is based on a set of 16 benchmark models from the area of active automata learning [214].

We target systems that can be modelled with a moderately large number of states, that is, with up to about fifty states. This restriction is necessary, because mutation analysis is a computationally intensive task for large systems. As noted in Section 2.5, this is also sufficient for many applications. Through abstraction it is possible to model many non-trivial reactive systems under this restriction. For instance, all Mealy machines used in our evaluation model real-world communication protocols and have at most 57 states. Since abstraction is generally required in learning-based verification, we do not consider this restriction to be severe.

Chapter Structure. The rest of this chapter is structured as follows. Section 4.2 presents our proposed process for test-suite generation involving mutation-coverage-based test-case selection. In Section 4.3, we introduce a mutation technique tailored towards active automata learning. Section 4.4 covers the evaluation of the proposed test-suite generation. In Section 4.5, we discuss efficiency in active automata learning in general, by presenting benchmarking experiments of various learning configurations. We provide a summary of our work in this area in Section 4.6 and conclude the chapter with a discussion of our findings in Section 4.7.

4.2 Test-Suite Generation

Aichernig et al. had shown previously "that by adding mutation testing to a random testing strategy approximately the same number of bugs were found with fewer test cases" [22]. Motivated by this, we developed a simple, yet effective test-suite generation technique. The test-suite generation has two parts, (1) generating a large set of test cases T and (2) selecting a subset $T_{\mathrm{sel}} \subset T$ to be executed on the SUL.

4.2.1 Test-Case Generation

The goal of the test-case generation is to achieve high coverage of the model under consideration combined with variability through random testing. Algorithm 4.1 implements this form of test-case generation based on a Mealy machine $\mathcal{M}_{\mathrm{h}} = \langle Q_{\mathrm{h}}, q_{0\mathrm{h}}, I, O, \lambda_{\mathrm{h}}, \delta_{\mathrm{h}} \rangle$ given as input. The test-case generation may start with a random walk through the model (Line 3 to Line 6) and then iterates two operations. First, a transition of the model is chosen randomly and a path leading to it is executed to cover that transition

Algorithm 4.1 The test-case generation algorithm

```
Input: $\mathcal{M}_\mathrm{h} = \langle Q_\mathrm{h}, q_{0_\mathrm{h}}, I, O, \lambda_\mathrm{h}, \delta_\mathrm{h} \rangle, p_\mathrm{retry}, p_\mathrm{stop}, \mathit{maxSteps}, l_\mathrm{infix},$
Output: $\mathit{test}$
 1: $\mathit{state} \leftarrow q_{0_\mathrm{h}}$
 2: $\mathit{test} \leftarrow \epsilon$
 3: if $\mathit{coinFlip}(0.5)$ then
 4:     $\mathit{test} \leftarrow \mathit{rSeq}(I, l_\mathrm{infix})$
 5:     $\mathit{state} \leftarrow \delta_\mathrm{h}(\mathit{state}, \mathit{test})$
 6: end if
 7: loop
 8:     $\mathit{rQ} \leftarrow \mathit{rSel}(Q_\mathrm{h})$                ▷ $(\mathit{rQ}, \mathit{rI})$ defines
 9:     $\mathit{rI} \leftarrow \mathit{rSel}(I)$                  ▷ a transition
10:     $p \leftarrow \mathit{path}(\mathit{state}, \mathit{rQ})$
11:     if $p \neq \mathit{None}$ then
12:         $\mathit{rSteps} \leftarrow \mathit{rSeq}(I, l_\mathrm{infix})$
13:         $\mathit{test} \leftarrow \mathit{test} \cdot p \cdot \mathit{rI} \cdot \mathit{rSteps}$
14:         $\mathit{state} \leftarrow \delta_\mathrm{h}(\delta_\mathrm{h}(\mathit{rQ}, \mathit{rI}), \mathit{rSteps})$
15:         if $|\mathit{test}| > \mathit{maxSteps}$ then
16:             break
17:         else if $\mathit{coinFlip}(p_\mathrm{stop})$ then
18:             break
19:         end if
20:     else if $\neg\mathit{coinFlip}(p_\mathrm{retry})$ then
21:         break
22:     end if
23: end loop
```

(Line 8 to Line 10). If the transition is not reachable, another target transition is chosen. Second, another short random walk is executed (Line 12). These two operations are repeated until a stopping criterion is reached.

Stopping. Test-case generation stops as soon as the test case has a length greater than a maximum number of steps $\mathit{maxSteps}$ (Line 15). Alternatively, it may also stop dependent on probabilities p_retry (Line 20) and p_stop (Line 17). The first one controls the probability of continuing in case a selected transition is not reachable, while the second one controls the probability of stopping prematurely.

Random Functions. Algorithm 4.1 uses the three random functions: *coinFlip*, *rSel*, and *rSeq*. These auxiliary functions have been introduced in Section 1.9. Here, they are used to control stopping, to create short random words, and to select random transitions in a Mealy machine.

The test-case generation is controlled by the stopping parameters and $l_\mathrm{infix} \in \mathbb{N}$, an upper bound on the number of random steps executed between visiting two transitions. The function *path* returns a path leading from the current state to another state. Currently, this is implemented via a breadth-first exploration given by Algorithm 4.2, but other approaches are possible as long as they satisfy $\mathit{path}(q, q') = \mathit{None}$ iff $\nexists \bar{i} \in I^* : \delta(q, \bar{i}) = q'$ and $\mathit{path}(q, q') = \bar{i} \in I^*$ such that $\delta(q, \bar{i}) = q'$, where $\mathit{None} \notin I$ denotes that no such path exists.

4.2.2 Test-Case Selection

To account for variations in the quality of randomly generated test cases, not all generated test cases are executed on the SUL, but rather a selected subset. This selection is based on coverage, for instance,

Algorithm 4.2 Breadth-first exploration implementing the function *path*

```
Input: $\mathcal{M}_\mathrm{h} = \langle Q_\mathrm{h}, q_{0\mathrm{h}}, I, O, \lambda_\mathrm{h}, \delta_\mathrm{h} \rangle$, arguments $q, q'$ of path
Output: either $p \in I^*$ such that $\delta(q, p) = q'$ or $None$
  $visited \leftarrow \{\}$
  $next \leftarrow emptyQ()$                              ▷ an empty queue of states to explore
  $next \leftarrow enqueue(next, \langle q, \epsilon \rangle)$              ▷ and traces leading to those states
  while $next \neq emptyQ()$ do
      $\langle q_c, p \rangle \leftarrow dequeue(next)$
      if $q_c \notin visited$ then                      ▷ current state not visited yet
          $visited \leftarrow visited \cup \{q_c\}$
          for all $i \in I$ do
              $q_n \leftarrow \delta(q_c, i)$
              if $q_n = q'$ then                        ▷ we found the target state
                  return $p \cdot i$
              end if
              $next \leftarrow enqueue(next, \langle q_n, p \cdot i \rangle)$
          end for
      end if
  end while
  return $None$                                       ▷ did not find a path to the target $q'$
```

transition coverage. For the following discussion, assume that a set of test cases T_sel of fixed size n_sel should be selected from a previously generated set T to cover elements from a set C. In a simple case, C can be instantiated to the set of all transitions by setting $C = Q_\mathrm{h} \times I$, as $(q, i) \in Q_\mathrm{h} \times I$ uniquely identifies a transition because of determinism. The selection comprises the following steps:

1. The coverage of single test cases is analysed. For that we associate each test case $t \in T$ with a set $C_t \subseteq C$ covered by t.

2. The actual selection has the objective of optimising the overall coverage of C. It is implemented by Algorithm 4.3. We greedily select test cases until either the upper bound n_sel is reached (Line 2 and Line 3), all elements in C are covered (second condition in Line 2), or we do not improve coverage (Line 4 and Line 5).

3. If n_sel test cases have not yet been selected, then further test cases are selected which individually achieve high coverage. For that $t \in T \setminus T_\mathrm{sel}$ are sorted in descending size of C_t and the first $n_\mathrm{sel} - |T_\mathrm{sel}|$ test cases are selected.

More sophisticated test-suite reduction/prioritisation strategies could have been used. Such strategies potentially have a high runtime, because the problem of selecting T_sel such that C is covered reduces to the set covering problem, which is known to be NP-complete [158]. Therefore, we use the efficient greedy selection.

4.2.3 Mutation-Based Selection

A particularly interesting selection criterion is selection based on mutation. The choice of this criterion is motivated by the fact that model-based mutation testing can effectively be combined with random testing [22]. Generally, in this fault-based test-case generation technique, known faults are injected into a model creating so-called mutants. Test cases are generated which distinguish these mutants from the original model. The execution of these test case covers the injected faults.

Algorithm 4.3 Coverage-based test-case selection

```
Input: $T$, $C$, $C_t$ for all $t \in T$, $n_{\text{sel}}$
Output: $T_{\text{sel}}$
1: $T_{\text{sel}} \leftarrow \emptyset$
2: while $|T_{\text{sel}}| < n_{\text{sel}} \wedge C \neq \emptyset$ do
3:     $t_{\text{opt}} \leftarrow \text{argmin}_{t \in T} |C \setminus C_t|$
4:     if $C \cap C_{t_{\text{opt}}} = \emptyset$ then
5:         break                                  ▷ no improvement
6:     end if
7:     $T_{\text{sel}} \leftarrow T_{\text{sel}} \cup \{t_{\text{opt}}\}$
8:     $C \leftarrow C \setminus C_{t_{\text{opt}}}$
9: end while
```

Thus, in our case we alter the hypothesis $\mathcal{M}_{\text{h}}$, creating a set of mutants $\mathcal{MS}_{\text{mut}}$. The objective is now to distinguish mutants from the hypothesis. We want to select test cases that show that mutants are observably different from the hypothesis. Hence, we can set $C = \mathcal{MS}_{\text{mut}}$ and $C_t = \{\mathcal{M}_{\text{mut}} \in \mathcal{MS}_{\text{mut}} \mid \lambda_{\text{h}}(t) \neq \lambda_{\text{mut}}(t)\}$ to achieve that.

Type of Mutation

The type of faults injected into a model is governed by mutation operators, which basically map a model to a set of mutated models (mutants). There is a variety of operators for programs [155] and also for finite-state machines [104]. As an example, consider a mutation operator *change output* which changes the output of each transition and thereby creates one mutant per transition. Since there is exactly one mutant that can be detected by executing each transition, selection based on such mutants is equivalent to selection with respect to transition coverage. Hence, mutation can simulate other coverage criteria. In fact, for our evaluation we implemented transition coverage via mutation.

Blindly using all available mutation operators may not be effective. Fault-based testing should rather target faults that are likely to occur in the considered application domain [236]. For this reason, we developed a family of mutation operators, called *split-state* operators, directly addressing active automata learning. We will discuss this kind of mutation in Section 4.3.

4.2.4 The Complete Testing Process

Now that we have discussed test-case generation, coverage-based selection, and mutation-based selection in particular, we want to give an overview of the proposed process for executing an equivalence query eq via mutation-based conformance testing. The data flow of the complete process involving test-suite generation and execution is shown in Figure 4.1.

The input to this process is a hypothesis $\mathcal{H}$, a learned intermediate Mealy machine model. From this model $\mathcal{H}$, we generate a set T of randomised test cases via Algorithm 4.1 and we generate a set $\mathcal{MS}_{\text{mut}}$ of mutants. Then, we analyse the mutation coverage C_t of each test case $t \in T$. We do that by executing each t and determining which mutants produce outputs different from the hypothesis' outputs. Next, the test suite for conformance testing is created by selecting a subset T_{sel} of T based on the coverage information C_t and by applying Algorithm 4.3. Finally, we execute all test cases of the test suite T_{sel}. A test case producing different outputs on the SUL than the outputs predicted by the hypothesis is a counterexample to equivalence. If such a counterexample is found, it is returned to the learning algorithm. If no counterexample is found, we assign a PASS verdict to the test suite. Such a PASS verdict translates to a positive answer to the equivalence query eq.

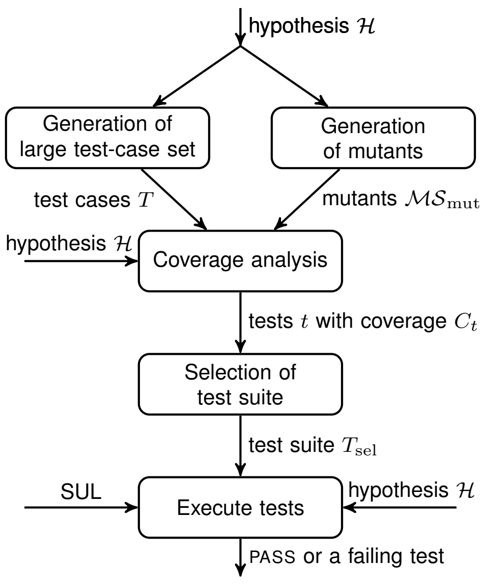


Figure 4.1: Data flow of test-suite generation and execution

4.3 Mutation for Learning

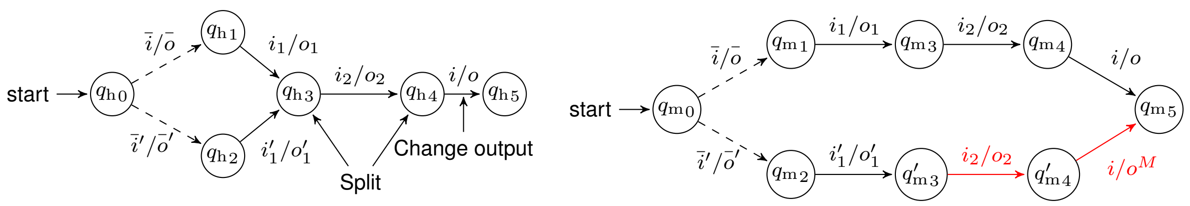
We gave a general overview of test-suite generation in the previous section. In the following, we present a mutation technique specifically targeting conformance testing in active automata learning. Hence, instantiating the process shown in Figure 4.1 with this technique yields an efficient testing method for learning. In the following, we will first introduce *split state* mutation operators, defining the form of mutants in our implementation of mutation testing. After that, we will present our implementation of mutant generation and optimised mutation analysis. We conclude the section with specifics affecting mutant generation and mutation analysis.

4.3.1 Split-State Mutation Operator Family

There are different ways to process counterexamples in the MAT framework, such as by adding all prefixes to the used data structures [37]. An alternative technique due to Rivest and Schapire [240] takes the "distinguishing power" of a counterexample into account. We described this technique in Section 2.3.1. Recall that its underlying idea is to decompose a counterexample into a prefix u, a single action a and a suffix v such that v is able to distinguish access sequences in the current hypothesis. In other words, the distinguishing sequence v shows that two access sequences, which were hypothesised to lead to the same state, actually lead to observably nonequivalent state; see also Example 2.6. This knowledge is then integrated into the learner's data structures.

Since this approach to counterexample processing is efficient, adaptations of it have been used in other learning algorithms such as the TTT algorithm [151]. This algorithm makes the effect of the decomposition explicit. It *splits* a state q reached by an access sequence derived from u and a. The splitting further involves (1) adding a new state q' reached by another access sequence derived from u and a (which originally led to q) and (2) adding sequences to the internal data structures which distinguish q from q'.

The development of the *split-state* family of mutation operators is motivated by the principle underlying the TTT and related algorithms. To create a single mutant of a Mealy machine $\mathcal{M}$, we take a pair (u, u') of access sequences of a state q of $\mathcal{M}$, add a new state q' to $\mathcal{M}$ and redirect u' to q'. Furthermore,



(a) A possible hypothesis model **(b)** A possible SUL/mutant

Figure 4.2: Demonstration of split-state mutations

we add transitions such that q' behaves the same as q except for a distinguishing sequence v. These steps split a state q to create a mutant. Example 4.2 illustrates this mutation technique.

Example 4.2 (Split State Mutation). A hypothesis produced by a learning algorithm may be of the form shown in Figure 4.2a. Note that not all edges are shown in the figure and dashed edges represent sequences of transitions. The access sequences $acc(q_{\mathrm{h}3})$ of $q_{\mathrm{h}3}$ include $\bar{i} \cdot i_1$ and $\bar{i}' \cdot i_1'$. A possible corresponding black-box SUL is shown in Figure 4.2b. In this case, the hypothesis incorrectly assumes that $\bar{i} \cdot i_1$ and $\bar{i}' \cdot i_1'$ lead to the same state. We can model a transformation from the hypothesis to the SUL by splitting $q_{\mathrm{h}3}$ and $q_{\mathrm{h}4}$ and changing the output produced in response to input i in the new state $q_{\mathrm{m}4}'$ from o to o^M, as indicated in Figure 4.2b. State $q_{\mathrm{h}4}$ has to be split as well to introduce a distinguishing sequence of length two while still maintaining determinism. A test case covering the mutation is $\bar{i}' \cdot i_1' \cdot i_2 \cdot i$.

Mutation operators usually capture fault models that represent certain types of faults [155]. Strictly speaking, a split-state mutation potentially "repairs" the current hypothesis to create a mutant that is closer to the true model. In slight abuse of terminology, we will use the term fault model when discussing split-state mutations rather than introducing a new term.

A mutant models a SUL containing two different states q and q' which are assumed to be equivalent by the hypothesis. By executing a test covering a mutant $\mathcal{M}_{\mathrm{mut}}$, we either find an actual counterexample to equivalence between SUL and hypothesis or prove that the SUL does not implement $\mathcal{M}_{\mathrm{mut}}$. Hence, it is possible to guarantee that the SUL possesses certain properties. This is similar to model-based mutation testing in general, where the absence of certain faults, those modelled by mutation operators, can be guaranteed [22].

Split state is a family of mutation operators as the effectiveness of the approach is influenced by several parameters, such that the instantiation of parameters can be considered a unique operator. The parameters are:

1. *Maximum number of sequences n_{acc}:* an upper bound on the number of mutated access sequences leading to a single state.

2. *Length of distinguishing sequences k:* for each splitting operation we create $|I|^k$ mutants, one for each input sequence of length k. This is necessary due to the input enabledness of Mealy machines. Note that introducing a distinguishing sequence of length k requires the creation of k new states. Coverage of all mutants generated with length k implies coverage of all mutants with length $l < k$.

3. *Split at prefix flag:* given an access sequence pair $(u \cdot a, u' \cdot a)$, redirecting a sequence $u' \cdot a$ from q to a new state q' usually amounts to changing $\delta_{\mathrm{h}}(\delta_{\mathrm{h}}(q_{0\mathrm{h}}, u'), a) = q$ to $\delta_{\mathrm{h}}(\delta_{\mathrm{h}}(q_{0\mathrm{h}}, u'), a) = q'$. However, if the other access sequence $u \cdot a$ is such that $\delta_{\mathrm{h}}(q_{0\mathrm{h}}, u') = \delta_{\mathrm{h}}(q_{0\mathrm{h}}, u)$, then we cannot simply redirect the transition labelled with input a in the state $\delta_{\mathrm{h}}(q_{0\mathrm{h}}, u')$. This is not possible, because it would introduce non-determinism. The *split at prefix* flag specifies whether the access sequence pair $(u \cdot a, u' \cdot a)$ is ignored or whether further states are added to enable redirecting $u' \cdot a$. We generally set it to **true**, that is, we add additional states to create mutants for such access sequence pairs.

Algorithm 4.4 Split-state mutant generation

```
Input: hypothesis $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$, $n_{\text{acc}}$, $k$
Output: set of mutants $Mut$
 1: $Mut \leftarrow \{\}$
 2: for $q \in Q$ do                    ▷ mutants for all unordered pairs of access sequences of $q$
 3:     for $\{s_1, s_2\} \in \{\{s_1, s_2\} \mid s_1, s_2 \in subset(acc(q), n_{\text{acc}})\}$ do
 4:         $Mut \leftarrow Mut \cup \textsc{Split}(s_1, s_2, \mathcal{M}, k) \cup \textsc{Split}(s_2, s_1, \mathcal{M}, k)$
 5:     end for
 6: end for

 7: function $\textsc{Split}(s_1, s_2, \langle Q, q_0, I, O, \delta, \lambda \rangle, k)$
 8:     if $\exists s : s_1 \cdot s = s_2$ then
 9:         return $\{\}$                                        ▷ $s_1$ prefix of $s_2$ ?
10:     else
11:         $eqSuf \leftarrow \text{argmax}_s\{|s| \mid \exists\, s_1', s_2' : s_1' \cdot s = s_1 \wedge s_2' \cdot s = s_2 \wedge \delta(s_1') = \delta(s_2')\}$
12:         $s_1' \cdot pI \cdot eqSuf \leftarrow s_1$          ▷ equivalent behaviour of $s_1$ and $s_2$ along $eqSuf$
13:         $Mutants \leftarrow \{\}$
14:         for all $distSeq \in I^k$ do      ▷ all possible distinguishing sequences of length $k$
15:             $q_{\text{sp}} \leftarrow \delta(s_1' \cdot pI)$                          ▷ $q_{\text{sp}}$ is split
16:             $q_{\text{pre}} \leftarrow \delta(s_1')$
17:             $\mathcal{M}' \leftarrow \langle Q', q_0, I, O, \delta', \lambda' \rangle$ s.t. $Q' = Q, \lambda' = \lambda$, and $\delta' = \delta \setminus \{(q_{\text{pre}}, pI, q_{\text{sp}})\}$
18:             $Q' \leftarrow Q' \cup \{q_s\}$                             ▷ copy original
19:             $\delta' \leftarrow \delta' \cup \{(q_{\text{pre}}, pI, q_s)\}$ for a new $q_s \notin Q'$
20:             $q_{\text{pre}_{\text{mut}}} \leftarrow q_s$
21:             $v \leftarrow eqSuf \cdot distSeq$
22:             $newQ \leftarrow \{\}$
23:             for $j \leftarrow 1$ to $|v|$ do
24:                 $q_{\text{orig}} \leftarrow \delta(s_1' \cdot pI \cdot v[< j])$
25:                 $q_{n\text{orig}} \leftarrow \delta(s_1' \cdot pI \cdot v[\leq j])$
26:                 if $j = |v|$ then             ▷ mutation transition back to original states
27:                     $\delta' \leftarrow \delta' \cup \{(q_{\text{pre}_{\text{mut}}}, v[j], q_{n\text{orig}})\}$
28:                     $\lambda' \leftarrow \lambda' \cup \{(q_{\text{pre}_{\text{mut}}}, v[j], o_{\text{mut}})\}$ s.t. $o_{\text{mut}} \neq \lambda(q_{\text{orig}}, v[j])$
29:                 else                        ▷ new states with indistinguishable behaviour
30:                     $Q' \leftarrow Q' \cup \{q_n\}$, for a new $q_n \notin Q'$
31:                     $newQ \leftarrow newQ \cup \{(q_n, q_{n\text{orig}})\}$
32:                     $\delta' \leftarrow \delta' \cup \{(q_{\text{pre}_{\text{mut}}}, v[j], q_n)\}$
33:                     $\lambda' \leftarrow \lambda' \cup \{(q_{\text{pre}_{\text{mut}}}, v[j], \lambda(q_{\text{orig}}, v[j]))\}$
34:                     $q_{\text{pre}_{\text{mut}}} \leftarrow q_n$
35:                 end if
36:             end for
37:             for all $(q, q_{\text{orig}}) \in newQ$ do
38:                 for all $i \in I$ do       ▷ copy original behaviour for transitions from new states
39:                     $\delta' \leftarrow \delta' \cup \{(q, i, \delta(q_{\text{orig}}, i))\}$
40:                     $\lambda' \leftarrow \lambda' \cup \{(q, i, \lambda(q_{\text{orig}}, i))\}$
41:                 end for
42:             end for
43:             $Mutants \leftarrow Mutants \cup \{\mathcal{M}'\}$                    ▷ add mutant
44:         end for
45:         return $Mutants$
46:     end if
47: end function
```

4.3.2 Implementation of Mutant Generation

The generation of mutants is implemented by Algorithm 4.4[2]. This algorithm requires the auxiliary function $\mathit{subset}(S, k)$, defined by $\mathit{subset}(S, k) = S'$ such that $|S'| = k$, $S' \subset S$ if $|S| > k$ and $S' = S$ otherwise. In the main loop of Algorithm 4.4 (Line 2 to Line 6), we process all unordered pairs $\{s_1, s_2\}$ of n_{acc} access sequences for a state q. Via the function SPLIT, we create mutants for these pairs. This function returns the empty set if one sequence is a prefix of the other (Line 8). The reason for this is that we would alter the behaviour for an input sequence v if we would redirect a prefix of v.

Otherwise, we compute the common suffix eqSuf of s_1 and s_2, along which both sequences visit the same states and execute the same inputs (Line 11). Then, we decompose s_1 into a prefix s_1', an input pI, and the suffix eqSuf. Mutants are created for all distinguishing sequences $\mathit{distSeq}$ (Line 14 to Line 44). Initially, mutants are copies of the original Mealy machine except for the state q_{sp} reached by $s_1' \cdot pI$, the state which is split, and the transition leading to this state (Line 16 to Line 19). We create a new state q_s for q_{sp} that is reached by pI. Furthermore, we create states along the sequence eqSuf (see *Split at prefix flag* above) to ensure determinism and along the distinguishing sequence $\mathit{distSeq}$ (Line 30 to Line 34). Finally, we mutate the last output corresponding to the last input of $\mathit{distSeq}$ (Line 28). All other transitions, added in Line 30 to Line 34 and in Line 38 to Line 41, ensure that the mutant produces the same outputs as the original hypothesis, if the distinguishing sequence is not executed.

In the implementation of $\mathit{acc}(q)$, we collect access sequences by performing a breadth-first exploration while traversing every state at most twice. Therefore, we may not find all n_{acc} access sequences for a state q. This strategy is motivated by performance considerations and the mutant sampling strategy *reduce to min*, which we commonly use.

4.3.3 Efficiency Considerations & Optimisation

While test-case generation can be implemented efficiently, mutation-based selection is generally computationally intensive. It is necessary to check which of the mutants is covered by each test case. Since the number of mutants may be as large as $|S| \cdot n_{\mathrm{acc}} \cdot |I|^k$, this may become a bottleneck. Consequently, cost reduction techniques for mutation [155] need to be considered.

Optimisation of Mutation Analysis

We reduce execution cost by avoiding the explicit creation of mutants. Essentially only the difference to the hypothesis is stored and executed. This optimisation is based on the following observation. In Algorithm 4.4, we can see that a mutant is uniquely identified by the triple $\langle \mathcal{M}, (q_{\mathrm{pre}}, pI), v\rangle$ where $\mathcal{M}$ is the original Mealy machine, (q_{pre}, pI) is a transition of $\mathcal{M}$ (Line 16) and $v = \mathit{eqSuf} \cdot \mathit{distSeq}$ is the sequence leading to the mutated output (Line 21). Hence, we basically check for coverage of a combination consisting of: (1) a sequence leading to transition (q_{pre}, pI), (2) the transition (q_{pre}, pI), and (3) the sequence v.

Given that insight, we implemented an efficient mutation analysis technique. Instead of explicitly creating all mutants, we rather generate all triples which implicitly describe all mutants. Let these triples be stored in a set IMut. We arrange this information in an NFA created by Algorithm 4.5. An NFA is a 5-tuple $\langle S, s_0, \Sigma, T, F\rangle$ with states S, an initial state $s_0 \in S$, an alphabet Σ, transitions $T \subseteq S \times \Sigma \times S$, and final states $F \subseteq S$. Since it is non-deterministic, there may be several transitions (s, e, s') for a source state s and a symbol e. Here, we set $\Sigma = I$, i.e. the symbols are the inputs of the original Mealy machine. Furthermore, let $S \subseteq Q \cup X \times \mathit{IMut}$ for a set of fresh unique symbols X. A state of the NFA is either a state of the original Mealy machine, or a new state corresponding to a mutant in IMut. The generation of the NFA by Algorithm 4.5 works as follows. Initially, the NFA has the same structure as the original Mealy machine but without outputs. Then, we add transitions and corresponding new

[2] In contrast to our published article [17], we use one-based indices in this thesis.

Algorithm 4.5 Creation of NFA representing sets of mutants

```
Input: hypothesis $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda\rangle$, mutants $IMut$
Output: nondeterministic finite automaton (NFA) $NMut$
 1: $NMut \leftarrow \langle S, s_0, I, T, F\rangle$ with $S \leftarrow Q$, $s_0 \leftarrow q_0$, $T \leftarrow \{(q, i, \delta(q, i))) \mid q \in Q, i \in I\}$, $F \leftarrow \{\}$
 2: for all $\mathcal{M}' = \langle \mathcal{M}, (q, pI), v\rangle \in IMut$ do
 3:     $s \leftarrow (x, \mathcal{M}')$ new state s.t. $s \notin S$, $S \leftarrow S \cup \{s\}$
 4:     $T \leftarrow T \cup \{(q, pI, s)\}$          ▷ $s$ corresponds to state that is split
 5:     for $j \leftarrow 1$ to $|v|$ do          ▷ add new states along sequence leading to mutation
 6:         $s' \leftarrow (x, \mathcal{M}')$ new state s.t. $s' \notin S$, $S \leftarrow S \cup \{s'\}$
 7:         $T \leftarrow T \cup \{(s, v[j], s')\}$
 8:         if $j = |v|$ then
 9:             $F \leftarrow F \cup \{s'\}$          ▷ final state corresponds to mutation
10:         end if
11:         $s \leftarrow s'$
12:     end for
13: end for
```

states for (q, pI) and v of all mutants (Line 3 to Line 11). The last state added for each of the mutants is a final state of the NFA (Line 9). Roughly speaking, a test case $test \in I^*$ covers a mutant $\mathcal{M}'$ if it reaches the corresponding final state $(x, \mathcal{M}')$. However, we need to account for the fact that mutants are deterministic, therefore it is not sufficient to check for reachability of final states via executing $test$ on the NFA.

The representation of mutants in an NFA is similar to the representation of mutants via mutation machines [166, 233]. Mutation machines are also non-deterministic finite-state machines, where sets of deterministic submachines define a fault domain. Likewise, mutation machines may be used to avoid the explicit enumeration of mutants.

The actual NFA-based mutation analysis is implemented by Algorithm 4.6. It implements a non-deterministic exploration of the NFA (Line 4 to Line 25), but if we enter an execution path corresponding to a mutant we temporarily block the corresponding transition (Line 9 to Line 13). Blocked transitions cannot be traversed (Line 8) and are stored in a mapping $blocked$, which maps from states to transitions. The exploration of the NFA works like the execution of several mutants in parallel. The blocking of transitions ensures that each mutant is executed at most once. Without this mechanism, we could execute multiple instances of a single mutant in parallel.

If we follow an execution path corresponding to a mutant, we update $blocked$ such that the newly reached state blocks the initially blocked transition (Line 10 and Line 11). If we are in a state blocking a transition and the current input $test[j]$ did not lead to a new state, then we left the execution path of the corresponding mutant. Consequently, we unblock the transition (Line 18 and Line 19). Finally, we identify covered mutants via final states. If a test case $test$ visits a final state $(x, \mathcal{M}')$, then $test$ covers the mutant $\mathcal{M}'$ (Line 23).

Mutant Sampling

Since the optimisation shown above does not solve the efficiency problem completely, mutant reduction techniques need to be considered as well. Jia and Harman identify four techniques to reduce the number of mutants [155]. We use two of them: *Selective Mutation* applies only a subset of effective mutation operators. In our case, we apply only one type of mutation operator, therefore we perform selective mutation. With *Mutant Sampling*, only a subset of mutants is randomly selected and analysed while the rest is discarded. Prior to sampling we collect all mutant triples $IMut$. Then, we apply three types of sampling strategies in our experiments:

Algorithm 4.6 Mutation-coverage analysis using the NFA-based mutant representation

```
Input: $test \in I^*, NMut = \langle S, s_0, \Sigma, T, F\rangle$
Output: Mutants $cMut$ covered by $test$
 1: $cMut \leftarrow \{\}$                                         ▷ covered mutants
 2: $state \leftarrow \{s_0\}$
 3: $blocked \leftarrow \{\}$
 4: for $j \leftarrow 1$ to $|test|$ do                            ▷ execute all steps of a test
 5:     $next \leftarrow \{\}$
 6:     for all $s \in state$ do
 7:         $n \leftarrow \{\}$                                     ▷ partial next state for state $s$
 8:         for all $t = (s, test[j], s') \in T$ s.t. $\nexists s'' : blocked(s'') = t$ do   ▷ follow unblocked $t$
 9:             if $s' = (x, \mathcal{M})$ then                     ▷ update $blocked$ if $t$ was added for mutant
10:                 if $\exists t' : (s, t') \in blocked$ then      ▷ if current state $s$ blocks a transition
11:                     $blocked \leftarrow (blocked \cup \{(s', blocked(s))\}) \setminus \{(s, blocked(s))\}$
12:                 else                                        ▷ block $t$ with target state $s'$
13:                     $blocked \leftarrow blocked \cup \{(s', t)\}$
14:                 end if
15:             end if
16:             $n \leftarrow n \cup \{s'\}$                        ▷ add target state $s'$ to next state
17:         end for
18:         if $n = \{\} \wedge \exists t : blocked(s) = t$ then    ▷ no trans. executed and blocked a transition
19:             $blocked \leftarrow blocked \setminus \{(s, blocked(s))\}$   ▷ unblock transition
20:         end if
21:         $next \leftarrow next \cup n$
22:     end for
23:     $cMut \leftarrow cMut \cup \{\mathcal{M}' \mid \exists x : (x, \mathcal{M}') \in next \wedge (x, \mathcal{M}') \in F\}$   ▷ newly covered mutants
24:     $state \leftarrow next$
25: end for
```

1. *Fraction:* this sampling strategy is parameterised by a real r. Given this parameter, we select $\frac{|IMut|}{2^r}$ of the mutants, where each mutant is selected uniformly at random from all mutants $IMut$. Hence, the resulting mutants are given by $subset(IMut, \frac{|IMut|}{2^r})$.

2. *Reduce to Minimum (redmin):* for this strategy, we partition $IMut$ into sets $IMut_q$ where q is the state that is split for creating mutants in $IMut_q$. A mutant $\mathcal{M}' = \langle \mathcal{M}, (q, pI), v\rangle$ with $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda\rangle$ is in $IMut_{\delta(q,pI)}$. After partitioning, we compute $m = \min_{q\in Q}(|IMut_q|)$, the minimum number of mutants in a set. Then, we select only m mutants from these sets and combine them again. The resulting mutants after sampling are given by $\bigcup_{q\in Q} subset(IMut_q, m)$. With this strategy we achieve a uniform coverage of all states.

3. *Reduce to Mean (redmean):* this strategy is similar to the last, but we select the average number of mutants in the sets $IMut_q$, i.e., $m = \frac{\sum_{q\in Q}|IMut_q|}{|Q|}$. It may be necessary to apply this strategy if there are states q with only one access sequence, that is, $|acc(q)| = 1$. In this case, there would be no access sequence pairs for q and consequently we would not create mutants for q such that *redmin* would discard all mutants.

4.3.4 Additional Implementation Details

Generally, the parameter choices, like the bound on the number of access sequences, the number of selected test cases, the sample size, etc. need to take the cost of executing test cases on the SUL into account. Thus, it is tradeoff between the cost of mutation analysis and testing, as a more extensive analysis can be assumed to produce more effective test cases which leads to fewer test-case executions.

In addition to mutant sampling, we identified two non-parameterised ways to reduce the number of mutants that we describe below.

1. *Mutation analysis of executed test cases:* we keep track of all test cases executed on the SUL. Prior to test-case selection, these test cases are examined to determine which mutants are covered by them. Covered mutants can be discarded, because we know for all executed test cases t and covered mutants $\mathcal{M}_{\mathrm{mut}}$ that $\lambda_{\mathrm{h}}(t) = \lambda_{\mathrm{sul}}(t)$ and $\lambda_{\mathrm{h}}(t) \neq \lambda_{\mathrm{mut}}(t)$. This implies $\lambda_{\mathrm{sul}}(t) \neq \lambda_{\mathrm{mut}}(t)$ which means that the covered mutants are not implemented by the SUL. Such a coverage check of executed test cases prevents unnecessary coverage of already covered mutants and reduces the number of mutants to be analysed. Furthermore, it takes the iterative nature of learning into account, as suggested by Howar et al. [144] in the context of equivalence testing. We analyse the coverage of test cases executed for equivalence queries and also of test cases executed during output queries. By analysing output-query coverage, we are able discard mutants whose mutations could be detected with a low number of interactions, consequently reducing computation time further.

2. *Adapting to learning algorithm:* by considering the specifics of a learning algorithm, the number of access sequences could be reduced. For instance in observation-table-based learning, as in L^* [37], it would be possible to create mutants only for access sequences stored in the rows of a table. We evaluated this approach for L^* combined with the counterexample processing by Rivest and Schapire [240]. This evaluation showed that stored access sequences are non-uniformly distributed across states. A few states are reached by a large number of sequences while other states are reached by only a low number of stored access sequences. Consequently, mutation analysis would concentrate on those states for which there are many access sequences. This turned out to be detrimental to test-case effectiveness, because a uniform coverage of all states should be preferred. Applying the *redmin* sampling strategy would achieve this. However, it would also discard too many mutants since it is likely that some states are reached by very few sequences in the observation table. As a result, we do not apply this approach, but ignore the internal data structures of learning algorithms.

4.4 Evaluation of Fault-Based Conformance Testing

In the following, we present the evaluation of two variations of our test-suite generation approach: one with transition-coverage-based selection and one with mutation-based selection applying split-state mutation. We will refer to test-case generation with transition-coverage-based selection as *transition coverage* and we will refer to the combination with mutation-based selection as *mutation*. We compare these two techniques to alternatives from the literature: the partial W-method [117] and the random version of the approach discussed in [254], available at [210]. We refer to the latter as *random L & Y*, as it applies an approach by Lee and Yannakakis [183] to compute adaptive distinguishing sequences. Note that this differs slightly from the setup in [112, 254] that also generates non-randomised test cases, to perform deterministic testing up to some bound. The random setup used in this evaluation for *random L & Y* is similar to the random W-method described in Section 2.4.2.

Our evaluation is based on our Java implementation of the presented test-suite generation approach that is available online [260]. In the learning experiments, we learn models of three implementations of TCP servers, four implementations of MQTT brokers, and nine implementations of TLS servers. The experimental results presented here have also been reported in our article on this topic [17]. The examined systems are summarised in Table 4.2. This table includes the number of states and the number of inputs of the true Mealy machine model and a short description of each system. For in-depth descriptions, we

Table 4.2: A short description of the systems examined in the evaluation

System	# States	# Inputs	Short Description
TCP servers			Models of TCP server/client implementations from three different vendors have been learned and analysed by Fiterău-Broştean et al. [112]. We simulated the server models available at [110].
Ubuntu	57	12	
Windows	38	13	
BSD	55	13	
MQTT brokers			Models of an MQTT [73] broker interacting with two clients that we learned with the *Two Clients with Retained Will* mapper described in Section 3.4.1 [263].
emqtt 1.0.2	18	9	
VerneMQ 0.12.5p4	17	9	
HBMQTT 0.7.1	17	9	
Mosquitto 1.4.9	18	9	
TLS servers			Models of TLS servers learned by de Ruiter and Poll [97], which are available at [98].
GnuTLS 3.3.8	16	11	
GnuTLS 3.3.12	9	12	
miTLS 0.1.3	6	8	
NSS 3.17.4	8	8	
OpenSSL 1.0.1j	11	7	
OpenSSL 1.0.1l	10	7	
OpenSSL 1.0.2	7	7	
RSA BSAFE C 4.0.4	9	8	
RSA BSAFE Java 6.1.1	6	8	

refer to the publications referenced in the table. All Mealy machines used in this evaluation are now also part of a set of benchmark models for conformance testing and learning [214].

4.4.1 Measurement Setup

To objectively evaluate randomised conformance testing, we investigate how many test-case executions are necessary to reliably learn models. For that, we estimate the probability of learning the correct model with a limited number of interactions with the SUL. Like in the Zulu challenge [88], only a limited number of test cases may be executed by each testing approach. We generally base the cost of learning in our discussion on the number of executed inputs rather than on the number of tests/resets. This decision follows from the observation that resets in the target application area, communication protocols, can be done fast (simply start a new session), whereas timeouts and quiescent behaviour cause long test durations [97, 263]; see also Section 3.4.5. Note that we take previously learned models as SULs. These models are given in a textual format specifying Mealy machines comprising the available inputs, outputs, transitions, and states. To simulate them, we built a test driver which produces outputs corresponding to given input sequences. Consequently, we still take a black-box view in which we interact with the simulated systems only via testing. As compared to testing of the actual systems, simulation of models allows fast test-case execution enabling a thorough evaluation.

We refer to one execution of a learning algorithm as a learning run. Such a learning run consists of several rounds, each concluded by an equivalence query that is carried out via conformance testing. To estimate the probability of learning the correct models with a given setup, we perform 50 learning runs and calculate the relative frequency of learning the correct model. In the following, we refer to such an execution of 50 learning runs as a single experiment. We deem learning reliable if the corresponding probability estimation is equal to one. Note that given the expected number of states of each system, we can efficiently determine whether the correct model has been learned, since the L^* algorithm guarantees that learned models are minimal with respect to the number of states [37]. In order to find a lower bound on the number of tests required by each method to work reliably, we bounded the number of test cases executed for each equivalence query and gradually increased this bound. Once all learning runs of an experiment succeeded we stopped this procedure. As in Section 4.2, we refer to this bound as n_{sel}, because it defines the number of test cases selected for execution. For learning with the partial W-method [117] we gradually increased the depth parameter d implemented in LearnLib [152] until we learned the correct model; see Line 3 of Algorithm 2.2 for the definition of d. Since the partial W-method does not use randomisation, we did not run repeated experiments and report the measurement results for the lowest possible depth setting.

As all algorithms can be run on standard notebooks, we will only exemplarily comment on runtime. For a fair comparison, we evaluated all conformance-testing approaches in combination with the same learning algorithm. We used L^* with Rivest and Schapire's counterexample-processing [240] implemented for Mealy machines by LearnLib 0.12 [152].

4.4.2 TCP Experiments

The number of tests and steps (input executions) required to reliably learn the different TCP servers are given in Table 4.3. In these experiments, we generated 200,000 test cases as a basis for the test-case selection of the approaches *mutation* and *transition coverage*. From these test cases, we selected the number of test cases n_{sel} given in the second row of Table 4.3 to perform each equivalence query. For *random L & Y* we generated the number of test cases given in the second row but we did not perform coverage-based selection from the test cases generated by this approach. For the partial W-method this row includes the depth-parameter value which determines the set of test cases to be executed for each equivalence query.

The test-case generation with Algorithm 4.1 has been performed with parameters $maxSteps = 60$, $p_{\mathrm{retry}} = 0.95$, $p_{\mathrm{stop}} = 0.05$, and $l_{\mathrm{infix}} = 6$. The chosen parameters for *mutation*-based selection are $k = 2$ (length of distinguishing sequence) and $n_{\mathrm{acc}} = 100$. Due to our implementation of acc (see Section 4.3), which returns access sequences that form at most one loop, setting n_{acc} to a value as large as 100 has the effect that no access sequences are discarded. Since we sample mutants afterwards, we set it to such a large value. We performed sampling by first applying the *redmin* sampling strategy and then *fraction* sampling with $r = 1$.

Note that we used the same setup for all TCP implementations. We only varied the bound n_{sel} on the number of equivalence tests. With that, we want to demonstrate that it is possible to learn several systems from the same application domain with similar setup. In practice, it is therefore possible to learn a first model with a conservatively chosen setup, by executing a large number of test cases. This model can be used to fine-tune parameter settings. Additional models with presumably similar structure could then be learned efficiently and reliably with the identified parameter settings.

As noted above, we executed 50 learning runs for each experiment. During one run, the number of tests for a single equivalence query (there may be several in one run), is bounded by the number given in Row 2[3] of Table 4.3. We collected data on the overall number of test cases executed for equivalence/membership queries as well as the overall number of inputs executed during those tests. The table shows these values averaged over all 50 runs of an experiment, where *eq* stands for equivalence queries and *mem* stands for membership queries. We also show further statistics with respect to the number of test steps executed for equivalence testing as we consider this to be the most important measure of performance. In addition to the complete information given in the table, we summarise the most important measures of performance in bar charts. Figure 4.3 provides an overview of the average number of required equivalence test steps and Figure 4.4 provides an overview of the required n_{sel}, which is the minimum number equivalence tests to learn reliably.

In Table 4.3, we see that the average number of tests and test steps required for membership queries is roughly the same for all techniques. This is what we expected, as the same learning algorithm is used in all cases, but the numbers shall demonstrate that techniques requiring less equivalence tests do not trade membership queries for equivalence tests. It can be considered a coincidence that learning with the

[3]Note that here we consider the table header to be Row 1.

Table 4.3: Performance measurements for learning TCP-server models

Ubuntu	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	3,500	5,000	2	44,000
mean # tests [eq.]	4,074	5,628	793,939	65,716
mean # steps [eq.]	154,551	350,678	7,958,026	703,315
median	151,758	334,147		643,785
Q1	141,497	322,584		588,320
Q3	164,089	346,430		761,456
min.	132,953	313,800		519,250
max.	226,119	612,634		1,338,526
mean # tests [mem.]	9,015	9,237	13,962	11,896
mean # steps [mem.]	120,195	128,276	147,166	138,340

BSD	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	1,500	2,000	5	58,000
mean # tests [eq.]	2,037	2,361	2,105,585,114	96,224
mean # steps [eq.]	91,089	152,301	30,435,822,650	1,069,553
median	86,031	147,513		1,002,069
Q1	77,142	139,952		857,912
Q3	99,824	156,330		1,225,874
min.	69,771	131,788		728,148
max.	152,133	270,108		1,985,223
mean # tests [mem.]	8,989	9,437	15,608	12,219
mean # steps [mem.]	132,126	141,588	170,416	146,893

Windows	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	3,500	2,500	2	40,000
mean # tests [eq.]	4,337	3,006	521,635	49,594
mean # steps [eq.]	144,074	178,985	4,597,896	514,458
median	131,750	167,658		492,830
Q1	125,873	158,452		455,614
Q3	158,914	195,210		555,076
min.	120,173	152,442		419,910
max.	210,253	298,169		839,380
mean # tests [mem.]	5,939	5,999	7,919	6,865
mean # steps [mem.]	65,137	63,547	67,834	68,563

partial W-method requires more membership queries while it also requires more equivalence tests. With this out of the way, we can concentrate on equivalence testing.

Figure 4.3 shows that *mutation* pays off in this experiment as it performs best for all different systems. It performs, for instance, significantly better than *transition coverage* for Ubuntu, which requires 2.27 times as many equivalence test steps on average. The average cost of test-case selection is approximately 324 seconds for *mutation* and 9 seconds for *transition coverage*. However, considering the large savings in actual test-case execution, *mutation* performs better. The small performance gain of *mutation* over *transition coverage* for Windows shows that our mutation analysis may not add value in some situations.

Considering the minimum required number of equivalence tests n_{sel} depicted in Figure 4.4, *mutation* and *transition coverage* show similar performance. Given that *transition coverage* executed more test steps suggests that it favours longer tests than *mutation*. Since both techniques require relatively few tests, they are applicable in setups, where reset operations are computationally expensive.

We also evaluated *random L & Y* with a middle sequence of expected length 3 for BSD and Ubuntu and length 4 for Windows (similarly to [112] where sequences of length 4 were used). For this setup, *random L & Y* requires significantly more steps and tests than both alternatives (see Figure 4.3 and Figure 4.4). There may be more suitable parameters, however, which might improve the performance of *random L & Y*. Moreover, the implementation also offers deterministic test-case generation for complete conformance testing up to some bound which may be beneficial as well. Nevertheless, the model structure of the TCP servers seems to be well-suited for our approach.

All randomised approaches outperformed the partial W-method. For instance for Ubuntu, *mutation* reduces the number of equivalence test steps by a factor of 51.5 on average (Row 4 of Table 4.3). Taking

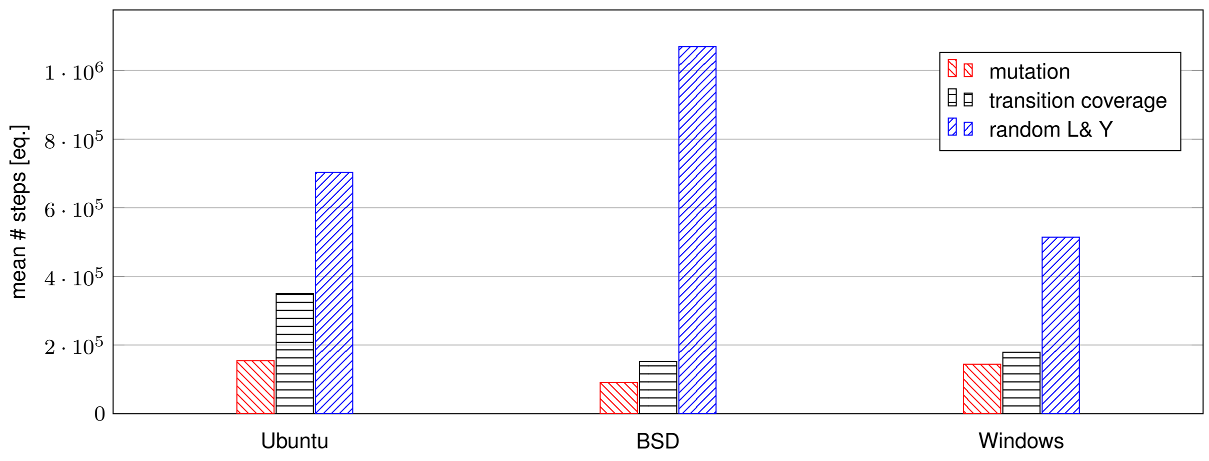


Figure 4.3: Average number of equivalence test steps required to reliably learn TCP models

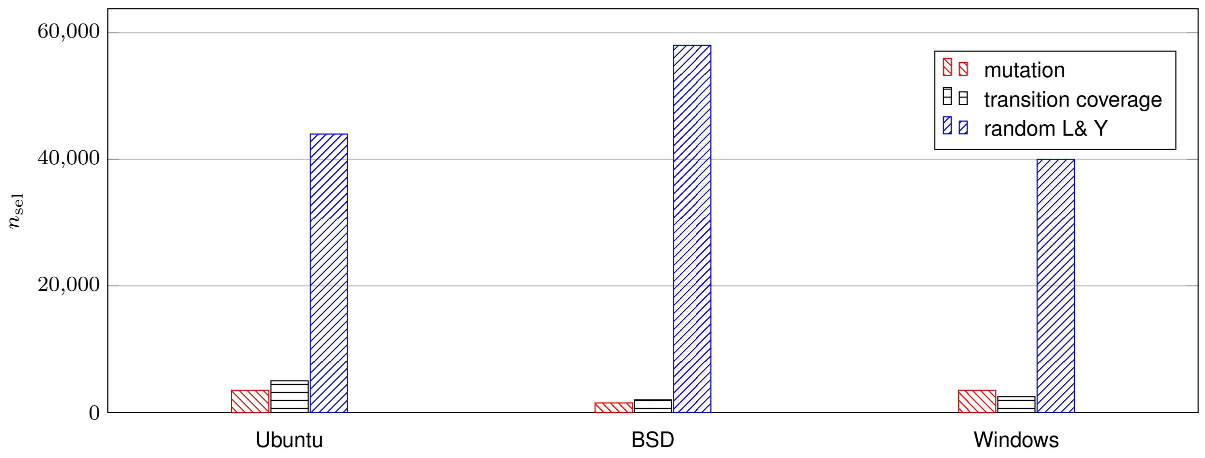


Figure 4.4: Minimum number of tests per equivalence query (n_{sel}) to reliably learn TCP models

the membership queries into account (Row 6 of Table 4.3), the overall cost of learning is reduced by a factor of 29.5. Comparing the average number of equivalence tests (Row 3 of Table 4.3) rather than the required test steps, we see an even larger reduction. The relative gap between equivalence tests shows a reduction by a factor of about 195 on average. This is an advantage of our approach as we can flexibly control test-case length which allows us to account for systems with expensive resets. The enormously large number of tests required by the partial W-method for BSD highlights a problem of complete conformance testing. Increasing the depth parameter causes an exponential growth of the number of tests. In practice, executing such a large number of test cases, as required to correctly learn the BSD model, would be infeasible. Completely random testing is not a viable choice in this case as well. A learning experiment with Ubuntu showed that it is practically infeasible to reliably learn correct models. We executed 1,000,000 random test cases for each equivalence query with a uniformly distributed length between 1 and 50 and learned correctly in only 4 out of 50 runs. For completely random test-case generation, we applied Algorithm 2.3.

Finally, we want to investigate the distribution of equivalence test steps. The gap between the first quartile $Q1$ and the third quartile $Q3$ is generally relatively small. Therefore, half of the runs of an experiment require roughly the same number of test steps. In other words, we see a uniform performance across runs. In general, the ratio between the minimum and the maximum number of steps is not very large as well. It is largest for *random L & Y* and BSD with a value of 2.73. This property of uniform performance of the randomised approaches can be explained by the following observations. Only a few tests are usually required to find a counterexample in the early phases of learning while the last

Table 4.4: Performance measurements for learning MQTT-broker models

emqtt	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	175	200	2	1,085
mean # tests [eq.]	253	253	72,308	1,664
mean # steps [eq.]	11,058	11,822	487,013	11,857
mean # tests [mem.]	1,647	1,603	1,808	1,683
mean # steps [mem.]	13,655	12,942	11,981	12,005
HBMQTT	**mutation**	**transition coverage**	**partial W-method**	**random L & Y**
n_{sel} / depth	200	325	2	4,350
mean # tests [eq.]	255	374	49,860	5,173
mean # steps [eq.]	10,986	17,212	334,298	35,781
mean # tests [mem.]	1,067	1,044	1,033	1,111
mean # steps [mem.]	9,116	8,257	6,133	7,513
Mosquitto	**mutation**	**transition coverage**	**partial W-method**	**random L & Y**
n_{sel} / depth	150	100	2	650
mean # tests [eq.]	187	132	59,939	1,036
mean # steps [eq.]	8,201	6,102	400,781	7,240
mean # tests [mem.]	1,309	1,372	1,355	1,378
mean # steps [mem.]	10,686	10,218	8,623	9,271
VerneMQ	**mutation**	**transition coverage**	**partial W-method**	**random L & Y**
n_{sel} / depth	125	125	2	1,200
mean # tests [eq.]	183	177	56,609	1,692
mean # steps [eq.]	7,983	8,245	375,263	11,947
mean # tests [mem.]	1,295	1,336	1,279	1,350
mean # steps [mem.]	10,410	10,063	7,985	9,193

equivalence queries require thorough testing. Put differently, a large portion of the state space can be explored with a small number of tests. Since there are no extreme outliers, we give only average numbers of equivalence test steps for the next case studies.

4.4.3 MQTT Experiments

The number of tests and test steps required to reliably learn models of four different MQTT brokers are given in Table 4.4. In order to perform these experiments, we used largely the same setup as for the TCP experiments, but generated only $50{,}000$ test cases as a basis for selection. Additionally, we decreased the maximum test-case length $maxSteps$ to 40 and changed the parameter r of the *fraction* sampling strategy to 0, to effectively only apply the *redmin* sampling strategy. As for TCP, we set $k = 2$ (length of distinguishing sequence introduced by split-state mutation). We applied *random L & Y* with a middle sequence of expected length 3. Figure 4.5 provides a graphical overview of the most important performance measure, the average number of equivalence test steps.

As before, Table 4.4 shows that the randomised approaches outperform the partial W-method. Considering test execution time in non-simulated setups highlights that applying such a randomised approach may save a significant amount of time. In the original learning setup for MQTT described in Chapter 3 [263], execution time was heavily affected by network communication. The execution of a single test step while learning VerneMQ models took 600 milliseconds (waiting for outputs from two clients with a timeout of 300 milliseconds per client). As a result, it would take approximately $(375{,}263 + 7{,}985) \cdot 0.6$ seconds ≈ 63.9 hours to learn a model of VerneMQ with the partial W-method. Applying the mutation-based approach *mutation* on the other hand would require $(7{,}983 + 10{,}410) \cdot 0.6$ seconds ≈ 3.1 hours while still learning reliably. Since these calculations take membership and equivalence queries into account, this gives us a reduction of overall learning time by a factor of about 20.6. We see varying but large reductions for all other MQTT brokers as well.

Figure 4.5 shows that all three of the randomised approaches perform similarly well with no clear winner. We can see in Table 4.4 that *random L & Y*, for instance, performs best for emqtt in terms

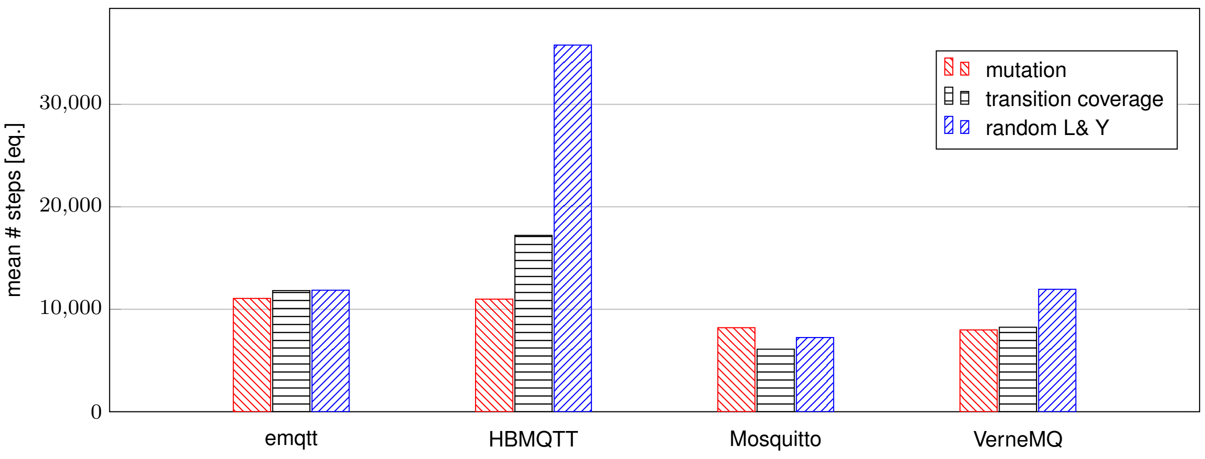


Figure 4.5: Average number of equivalence test steps required to reliably learn MQTT models

of executed test steps, if we also take membership queries into account (Row 6 of Table 4.4). The *transition coverage* approach requires the least number of steps for the Mosquitto model, but performs only slightly better than *random L & Y*. However, *mutation* also performs well for this example. We assume that our mutation-based approach produces good results in general, but as demonstrated by the MQTT experiments, it is not the best-performing solution in all circumstances.

Figure 4.6 shows how reliably each of the three approaches learned the emqtt model depending on the number of executed test steps. Additionally, it also provides a comparison to completely random test-case generation implemented by Algorithm 2.3. Each input of the random test cases is chosen uniformly at random from all available inputs and the test-case length is uniformly distributed between 1 and 50. As noted at the beginning of this section, we say that a model is learned reliably if it is learned correctly with a high probability. We estimate the probability of learning correctly by computing the relative number of successful learning runs, which are runs in which the correct model was learned. This estimation is denoted by $\hat{p}_{\text{correct}}$ and labels the y-axis in Figure 4.6. The x-axis of Figure 4.6 gives the number of executed equivalence test steps. The leftmost data point of the black line with +-markers, for instance, lies at $x = 529.66$ and $y = 0.12$ and represents the result of a *transition-coverage* learning experiment. It denotes that *transition coverage* learned correctly in 6 out of 50 learning runs while executing 530 equivalence test steps on average.

We see in Figure 4.6 that the reliability of *random L& Y* and *transition coverage* grows more smoothly than the reliability of *mutation*. Although allowing for more equivalence test steps generally increases reliability of learning, we also see slight drops for *mutation*. The reason for this behaviour is that we are able to cover only a small portion of the mutants with a low number of test steps/cases such that we may select test cases covering irrelevant mutants. Covering all transitions on the other hand is simpler and *random L & Y* follows a rather different approach. However, if we allow for sufficiently many test cases to be selected and executed, these drops can be expected to disappear. Despite these drops, *mutation* requires slightly less test steps to learn reliably. Furthermore, we see that random testing may be applied successfully, but it is less reliable at producing correct results.

4.4.4 TLS Experiments

For the last case study, we learn models of nine TLS-server implementations [98]. The models were originally learned by de Ruiter and Poll [97] while taking domain-specific knowledge into account. They stopped executing a test case once the connection between the TLS server and the test harness had been closed. This is reflected in the structure of the models such that we may use shorter tests. Therefore, we set the following parameters for test-case generation: $\mathit{maxSteps} = 20$, $p_{\text{retry}} = 0.95$,

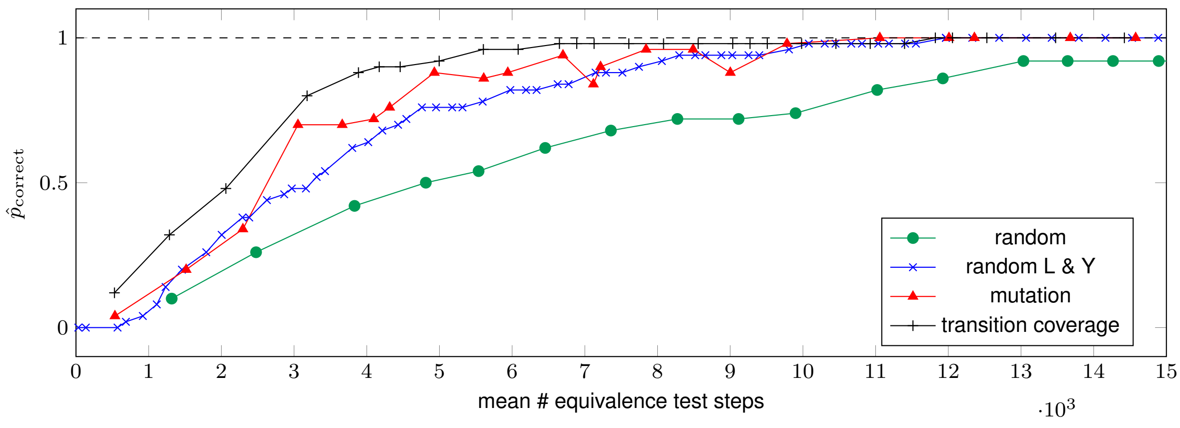


Figure 4.6: Reliability of learning the correct emqtt-broker model with a limited number equivalence test steps

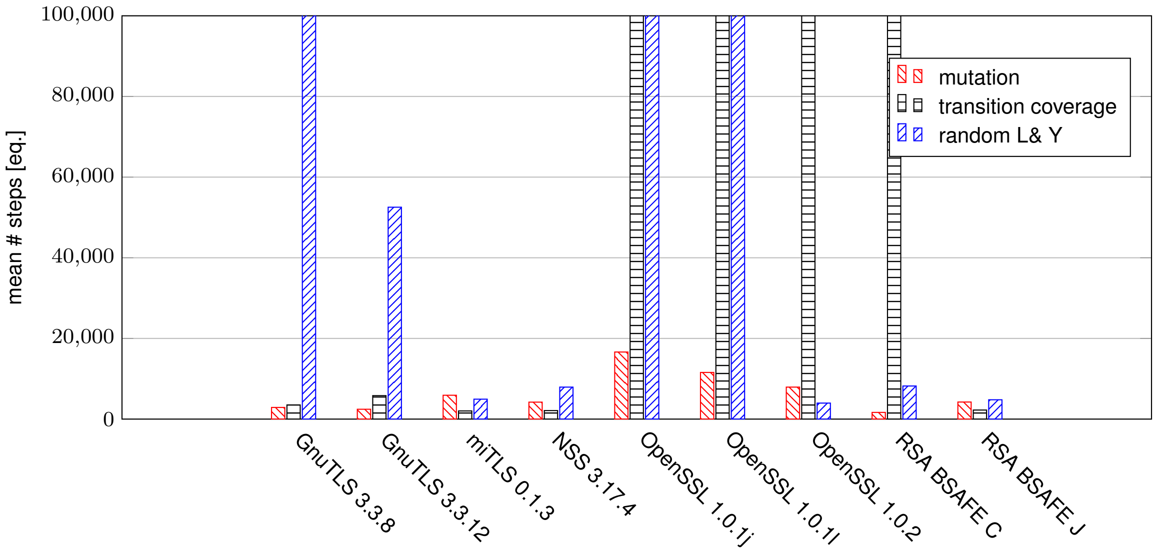


Figure 4.7: Average number of equivalence test steps required to reliably learn TLS models

$p_{\text{stop}} = 0.05$, and $l_{\text{infix}} = 3$. Compared to the previous experiments, we reduced the bound on test-case length and we also reduced the length of the purely random sequences created by Algorithm 4.1. Since there are nine implementations with varying structure (see the number of states in Table 4.2), we decided to perform a thorough coverage analysis by generating 300,000 test cases as a basis for test-case selection. We used the same mutation parameters as before but a different sampling strategy. Some of the states of the considered models were reached by only one access sequence such that no mutants would be produced for these states. As a result, the *redmin* strategy would discard all mutants, including those mutants produced for states with multiple access sequences. Instead, we applied the *redmean* strategy and *fraction* sampling with $r = 1$.

Since we have seen that membership queries are not affected by different approaches to equivalence testing, we now provide only measurement results for equivalence testing. These results are listed in Table 4.5. Like for all other models, we provide an overview of the average number of executed equivalence test steps. This overview is shown in Figure 4.7. The first issue to notice is that for *transition coverage* and the OpenSSL models, we would need to set the equivalence test bound n_{sel} to a value larger than 10,000. For this reason, we stopped at this point and deemed learning infeasible with this setup. The same holds for *random L& Y* and GnuTLS 3.3.8 where we performed experiments with up to 20,000 tests, because the generated test cases were shorter than the test cases generated by Algorithm 4.1. The *mutation* approach on the other hand was successful for all models.

Table 4.5: Performance measurements for learning TLS-server models

GnuTLS 3.3.8	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	100	200	3	$>$ 20,000
mean # tests [eq.]	182	283	709,845	-
mean # steps [eq.]	2,888	3,513	4,979,346	-
GnuTLS 3.3.12	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	100	300	1	7,600
mean # tests [eq.]	141	333	1,354	8,776
mean # steps [eq.]	2,441	5,819	5,815	52,536
miTLS 0.1.3	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	300	100	1	700
mean # tests [eq.]	347	122	1,017	884
mean # steps [eq.]	5,913	2,027	4,508	4,939
NSS 3.17.4	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	200	100	1	1,000
mean # tests [eq.]	274	138	1,428	1,357
mean # steps [eq.]	4,206	2,115	5,970	7,932
OpenSSL 1.0.1j	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	1,100	$>$ 10,000	2	12,400
mean # tests [eq.]	1,209	-	13,853	15,392
mean # steps [eq.]	16,635	-	78,615	100,332
OpenSSL 1.0.1l	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	800	$>$ 10,000	2	15,300
mean # tests [eq.]	864	-	12,666	18,332
mean # steps [eq.]	11,566	-	68,524	115,122
OpenSSL 1.0.2	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	500	$>$ 10,000	1	500
mean # tests [eq.]	529	-	916	690
mean # steps [eq.]	7,934	-	3,621	3,970
RSA BSAFE for C 4.0.4	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	100	5,000	1	1,100
mean # tests [eq.]	125	5,831	980	1,369
mean # steps [eq.]	1,656	106,563	4,188	8,209
RSA BSAFE for Java 6.1.1	mutation	transition coverage	partial W-method	random L & Y
n_{sel} / depth	200	100	1	600
mean # tests [eq.]	254	137	960	873
mean # steps [eq.]	4,235	2,247	3,877	4,793

In Table 4.5, we notice better performance of the partial W-method. It even requires the lowest number of test steps for OpenSSL 1.0.2. This stands in stark contrast to previous observations, but is simply caused by the low depth-parameter value required for learning TLS models. It actually highlights the effect of the depth parameter. To correctly learn the GnuTLS 3.3.8 model, the depth parameter needs to be at least 3. For this model, *mutation* needs three orders of magnitude fewer test steps. If we were to choose 2 as depth-parameter value for all other models, to be on the "safe side", the performance of the partial W-method would drastically drop. Note that we usually do not know the required depth.

Comparing the randomised approaches in Figure 4.7, we see that *mutation* generally performs well. It is not the best performing solution in all cases, though. We made the same observation above for MQTT as well. Hence, *mutation* may be better suited to changing environments. In comparison to the other approaches, it shows the worst performance for miTLS, for which it requires three times as many test steps as *transition coverage*. However, it performs best for other models, like for OpenSSL 1.0.1j. Note that it does not show extreme outliers as well. The different versions of OpenSSL, for example, cannot be learned efficiently with *transition coverage*.

It should be noted that the non-randomised version of *random L & Y* discussed in [254] can be assumed to perform best for the models that can be learned by setting the depth parameter to 1. The reason for this is that *random L & Y* implements an effective method for computing distinguishing sequences. Moreover, the partial W-method already performs well for these models.

4.4.5 Discussion, Limitations & Threats to Validity

Effectiveness of Mutation-based Selection. The measurements shown and discussed above suggest that our mutation-based test-case selection pays off. *Mutation* performed very well in the TCP case study, but did not perform best in some other cases. Still, it generally showed good performance in comparison to the other randomised approaches. There are, for instance, systems for which *transition coverage* performs slightly better. In such cases, the mutation-based selection apparently does not add value. It should be noted that a crucial aspect of *mutation* is that it uses Algorithm 4.1 for test-case generation. Initially, we generated test cases purely randomly. These tests, however, would fail to reach most of the mutations. The coverage-directed generation of Algorithm 4.1 mitigates this issue. A potential further improvement would be to perform a mutation-directed, but still randomised, test-case generation. Preliminary experiments with such an approach required very high runtimes for test-case generation, therefore we did not explore it any further. Algorithm 4.1 represents a good tradeoff between test-case generation efficiency and test-case effectiveness.

Application Domain. Our evaluation is limited to variations of three types of systems, therefore it may not generalise to other systems, especially from other application domains. All considered are implementations of communication protocols. Since communication protocols are the main focus of this thesis, we decided to thoroughly evaluate our approach with respect to such systems. We evaluate additional configurations of test-based learning with respect to efficiency in the domain of communication protocols in Section 4.5, providing further evidence for the effectiveness of mutation-based test-case selection.

Guarantees. We compared *mutation* with the partial W-method, which provides certain completeness guarantees. The total cost of learning in our setting is given by the average number of test steps combined for equivalence and membership queries. *Mutation* improves upon the partial W-method with respect to this measure by at least one order of magnitude for all MQTT and TCP models. It may be argued that the comparison is unfair because the partial W-method provides guarantees while *mutation* aims at optimising coverage. In the considered setup, however, *mutation* provides guarantees as well. Since the automata learned with L^* are minimal [37], we can state: if we learn a model, then it is either correct or the SUL has more states than the learned model. The (partial) W-method performed with depth d provides the stronger guarantee: if we learn a model with k states, then it is either correct or the SUL has more than $k + d$ states [117].

Given the difference in guarantees, we decided to perform another type of experiment that we want to discuss only briefly. In this type of experiment, we took the maximum number of steps required by *mutation* as a basis and we bounded the number of equivalence test steps for the partial W-method by this number. As an illustrative example consider learning the Ubuntu TCP-server model. The *mutation* approach requires a maximum of 226,119 equivalence test steps to learn reliably. Equivalence testing with the partial W-method with depth 2 while additionally bounding the number of steps by 226,119, resulted in learning of a model with 27 states. Hence, the guarantee provided by that is that the model is either correct or the system has more than 29 states. Strictly speaking, the guarantee would actually be weaker, because we did not execute all test cases prescribed by the partial W-method. If we learn with *mutation*, we learn a model with 57, thus providing us with a lower bound of 57 on the number of states of the true model. Hence, conditioned on allowed test-execution time, learning with *mutation* is able to provide stronger guarantees.

Parameter Settings. An advantage of the randomised approaches is that they can be controlled more flexibly through parameters. However, a large number of parameters also increases the difficulty of finding suitable parameter settings. This holds for all performed experiments. There may be better parameters for *mutation* as well as for *random L & Y*. In general, we tried to find parameters that would work for one of the MQTT/TCP/TLS models and used these settings for all the other models as well. Since we developed the *mutation* technique, the learning performance in our evaluation may be biased. Put differently, it is likely that we are able to find well-suited parameters for our approach but not for *random L & Y*. For this reason, performance gains of *mutation* over *random L & Y* may be smaller in general.

Another risk related to the large parameter space of *mutation* is that it may be possible that only a small portion of the parameter settings actually produces good results. To limit the risk that we found good parameters by chance, we used the same parameter settings for all systems of a certain type. The TLS case study shows that the same parameter settings may work nicely for various different systems. Still, finding parameter settings for *mutation* is likely to be the most difficult.

Variability. Another benefit of randomised approaches is that randomness introduces variability which may help. For the BSD case study, we needed to set the depth of the partial W-method to 5. In principle, this implies for *mutation* that mutants should be created with distinguishing sequences of length $k = 5$. But we set $k = 2$ and still learned successfully. This may be explained by the fact that if we cover a large number of mutants with $k = 2$, we will with high probability cover mutants with $k > 2$ as well. Additionally, variability in tests may help to explore the search space more thoroughly. Completely random testing did not perform well, though. Without any form of directed testing, we fail to reach relevant portions of the search space.

System Size and Computation Time. A shortcoming of *mutation* is that mutation analysis is computationally intensive although we spend effort optimising it. Comparing the test-case generation durations for the different protocols, we see an increase with system size. Mutation-based test-case generation requires on average 27 seconds during a complete learning run of OpenSSL $1.0.1$j and it requires 45 seconds for emqtt. Due the substantially larger state space, it takes 324 seconds to generate test cases for the Ubuntu TCP-server. Therefore, we target moderately-sized systems with the mutation-based approach.

As *random L & Y* generates test cases more efficiently, it can be applied for larger systems as well. It has, for instance, been used to learn a system with more than $3{,}000$ states [254]. Applying *mutation* to systems with significantly more than 100 states would likely not pay off. We would have to perform aggressive mutant sampling which might negatively impact the effectiveness of mutation-based selection. Note that *transition coverage* does not suffer from this limitation. We were able to learn a model with more than $6{,}000$ states by conformance testing via *transition coverage*. Chapter 10 presents a case study on that.

In general, we did not evaluate the influence of mutant sampling. We have observed during development that sampling is detrimental to test-case effectiveness if we discard too many mutants. However, a thorough evaluation would improve our understanding of the effects and limitations of sampling in the context of active automata learning. There are various works studying the effects of mutant sampling on traditional mutation testing [155].

4.5 Benchmarking Active Automata Learning

Declaration of Sources

As indicated at the beginning of this chapter, this section is based on work performed by Felix Wallner for his bachelor's thesis, which was co-supervised by me [288].

Table 4.6: Evaluated learning and testing algorithms

Learning algorithm	Testing algorithm
L^* [37, 246]	W-method [83, 277]
RS [240]	partial W-method [117]
KV [160]	random words (see Algorithm 2.3)
TTT [151]	random walks (see Algorithm 2.4)
	mutation (see Section 4.4)
	transition coverage (see Section 4.4)
	random Wp-method (see Section 2.4.2)

Section 4.4 focused on the evaluation of our mutation-based test-case generation, comparing it to different testing techniques. To enable a fair comparison, we used the same learning algorithm in combination with each of the four testing techniques that we applied. We generally measured efficiency in terms of the number of test steps required to perform equivalence queries via testing. Additionally, we also measured the number of test steps for output queries and the number of tests/resets performed for equivalence and output queries.

In this section, we present similar measurements performed by Felix Wallner for his bachelor's thesis [288], but we compare combinations of learning algorithms and testing algorithms. The goal of these measurements is to evaluate the overall performance of learning. This evaluation, for instance, examines the performance of L^* combined with random-walks-based testing implemented by Algorithm 2.4. As in Section 4.4, we "re-learn" known models of network protocols for that. Thus, we treat these models as black boxes and simulate them to generate outputs in response to input sequences. The setup is also mostly the same, but we perform ten learning runs rather than 50 in each experiment and compute statistics from that. We lowered the number of learning runs to be able to perform a larger number of experiments in reasonable time. Since we are interested in the overall performance of test-based learning, we mainly consider the combined number of test steps required for equivalence queries and output queries.

4.5.1 Measurement Setup

Selection of Algorithms. Altogether, we evaluated all combinations of four learning algorithms and seven testing techniques, that is, we evaluated 28 learner-tester combinations. The learning algorithms are listed in the first column of Table 4.6 and the testing techniques are listed in the second column of Table 4.6. We have chosen the *random Wp-method* over *random L&Y*, because it is readily available as equivalence-query implementation in LearnLib [152]. Both techniques produce test cases of similar shape, but *random L & Y* tends to perform better. Note that the W-method creates test suites that are generally larger than the partial W-method. Individual equivalence queries using the partial W-method are therefore more efficient. However, the partial W-method and the W-method may find different counterexamples, therefore intermediate hypotheses may be different as well. For this reason, we included both testing algorithms in our evaluation. In contrast to Section 4.4, we use LearnLib 0.14 instead of LearnLib 0.12, thus results may differ from Section 4.4.

Benchmark Models. In this section, we consider a subset of the benchmark models from the automata-learning benchmark models collected by the Radboud University Nijmegen [214][4]. In particular, we use all six TCP models, including both server and client models of the TCP stacks of Ubuntu, Windows, and BSD, learned by Fiterău-Broştean et al. [112]. We consider all 32 MQTT models, created in our learning-based testing case study presented in Chapter 3. Finally, we also consider a simple coffee machine that is similar to a model used by Steffen et al. [258]. The evaluation in this section does not include experiments

[4]available via `http://automata.cs.ru.nl/`, accessed on November 4, 2019

with the TLS models learned by de Ruiter and Poll [97], because like most of the MQTT models, the TLS models have a low number of states. In order to keep the imbalance between large and small models low, we decided to not include the TLS models. The experimental results presented in the following are thus based on learning experiments with 39 benchmark models.

Search for Required Number of Tests. Like in Section 4.4, we search for the minimum number of test cases that need to be executed during individual equivalence queries to learn reliably using randomised testing techniques. As before, we denote this number by n_{sel}. Moreover, we consider learning to be reliable if all ten learning runs in a learning experiment are successful. Since the relative number of small models is larger than in Section 4.4, we perform a fine-grained binary search for n_{sel}, instead of a coarse-grained linear search.

Configuration of Testing Techniques. We apply the same configuration of every testing technique for all considered models. The configurations have been chosen to enable learning of system with up to approximately 50 states. For instance, we configured random-words-based testing such that all generated test cases have a length between 10 and 50. Note that we apply more aggressive mutant sampling than in Section 4.4, because we perform more experiments. The parameter configurations are as follows.

random words: minimum length $l_{\min} = 10$ and maximum length $l_{\max} = 50$; see also Algorithm 2.3.

random walks: test stop probability $p_{\mathrm{stop}} = \frac{1}{30}$; see also Algorithm 2.4. This setting ensures that the expected length of random walks is the same as of random words.

random Wp-method: we set the minimal length of the middle sequence to 0 and the expected length to 4; see also the discussion of the random W-method in Section 2.4.2.

transition coverage: $maxSteps = 50$, $p_{\mathrm{retry}} = \frac{29}{30}$, $p_{\mathrm{stop}} = \frac{1}{30}$, and $l_{\mathrm{infix}} = 4$; for a description of the parameters see Section 4.2 and Section 4.4. Note that we have chosen p_{stop} to be the same value as p_{stop} of random walks.

mutation: $maxSteps = 50$, $p_{\mathrm{retry}} = \frac{29}{30}$, $p_{\mathrm{stop}} = \frac{1}{30}$, $l_{\mathrm{infix}} = 4$, $k = 2$ (length of distinguishing sequence), and $n_{\mathrm{acc}} = 100$. We applied the *redmin* mutant sampling strategy, then applied *fraction* sampling with $r = 1$ and finally sampled 10,000 of the remaining mutants, unless there were less than 10,000 mutants remaining after *fraction* sampling.

The only parameter of the deterministic algorithms, the *W-method* and the *partial W-method*, is the depth parameter. We increased this parameter linearly until we learned correctly, like in Section 4.4. For *transition coverage* and *mutation*, we created $\min(100{,}000, n_{\mathrm{sel}} \cdot 100)$ test cases using Algorithm 4.1 and selected n_{sel} of these test cases based on coverage. In the remainder of this section, we write testing techniques in italics like in Section 4.4.

4.5.2 Measurement Results

Altogether we performed 39 learning experiments with each of the 28 learner-tester combinations. We present selected results from these experiments in the following, focusing on the number of test steps required for both equivalence queries and output queries. In particular, we consider the maximum and mean number of test steps required to learn reliably. Due to the large amount of learning experiments, we present aggregated results for learner-tester combinations in (1) cactus plots and (2) bar plots. Additional information and the complete results can be found in Felix Wallner's bachelor's thesis and the accompanying supplementary material [288].

The cactus plots show how many experiments can be finished successfully, such that learning is reliable, given a maximum number of test steps. The bar plots show two different scores, s_1 and s_2,

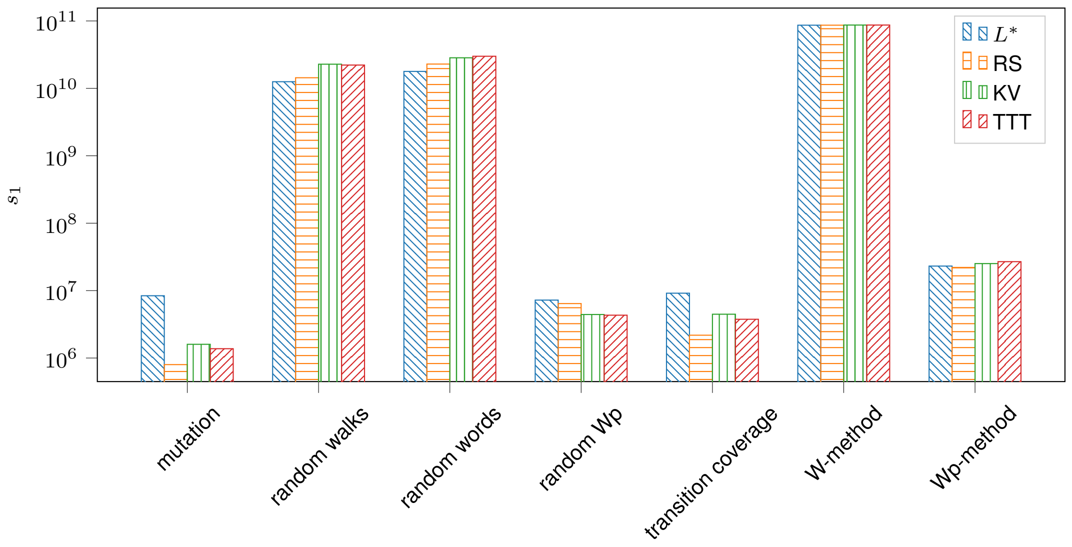


Figure 4.8: The score s_1 computed over all experiments for all learner-tester combinations, grouped by testing technique

computed for the learner-tester combinations lt. The actual scores are not important, but they allow for comparisons, where a lower value means better performance. The scores are given by

$$s_1(lt) = \sum_{e \in E} \mathit{meanAllSteps}(lt, e) \text{ and}$$

$$s_2(lt) = \sum_{e \in E} \frac{\mathit{meanAllSteps}(lt, e)}{\max_{lt' \in LT} \mathit{meanAllSteps}(lt', e)},$$

where E is the set of considered experiments, LT is the set of all learner-tester combinations, and $\mathit{meanAllSteps}(lt, e)$ returns the mean number of steps to reliably learn with the combination lt in the experiment e. The first score $s_1(lt)$ simply sums up the average number of test steps required in all experiments, whereas $s_2(lt)$ is normalised, through dividing by the worst-performing combination of each experiment. Hence, s_1 allows to analyse which combinations perform best, when learning all 39 models consecutively and under the assumption that test steps require the same amount of time in each experiment. The normalised score s_2 accounts for the large variation in terms of model complexity across the different experiments. Normalisation ensures that individual performance outliers do not severely affect the overall score of a learner-tester combination. As information about outliers is useful, it is represented in the cactus plots.

Categories. Certain behavioural aspects of communication-protocol models may favour a particular learner-tester combination, but other aspects may favour different combinations. For this reason, we grouped the benchmark models into categories based on the following properties:

- *small:* a model is *small*, if it has less than or equal to 15 states
- *large:* a model is *large*, if it has more than 15 states
- *sink-state:* a model satisfies the property *sink-state*, if there exists a (sink) state q such that all outgoing transitions from q reach q
- *strongly-connected:* a model satisfies the property *strongly-connected*, if its underlying directed graph is strongly connected, that is, for each ordered pair of nodes exists a directed path between these nodes

For instance, we examined which combinations perform best for small models that have a sink state.

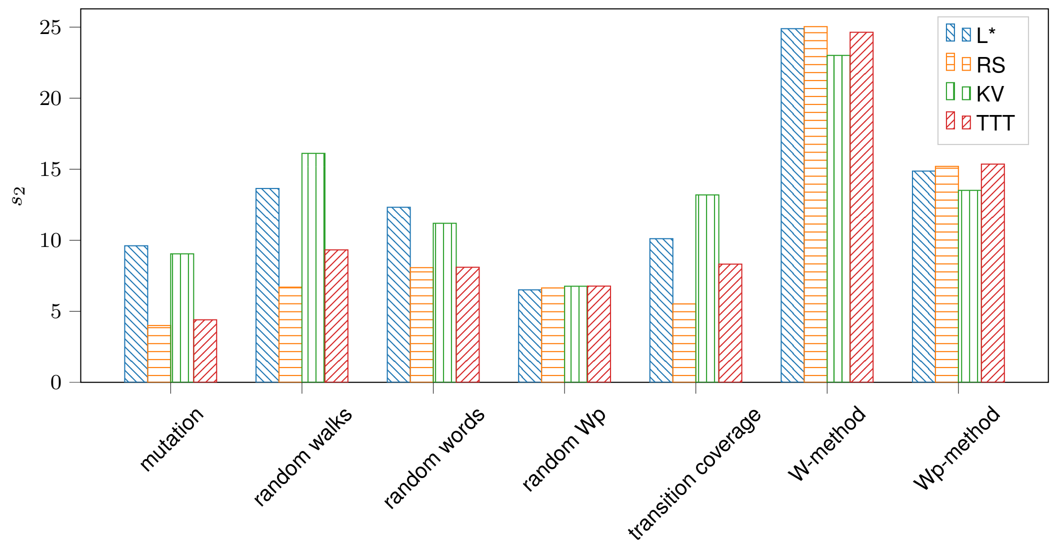


Figure 4.9: The score s_2 computed over all experiments for all learner-tester combinations, grouped by testing technique

Overview

First, we want to provide a rough overview. Figure 4.8 shows the score $s_1(lt)$ for each learner-tester combinations computed over all experiments. Due to large variations in the required test steps it uses logarithmic scale. Figure 4.9 shows the normalised score $s_2(lt)$. We see a similar picture as in Section 4.4. *Mutation*, *transition coverage*, and *random Wp* perform well in comparison to other techniques. In Figure 4.8, we can observe that the relative gap between *mutation* and the worst-performing techniques is very large. This is caused by a few outliers. In particular, the TCP server models required a very large number of test steps for *random walks* and *random words* to learn reliably. For this reason, we see a smaller gap between those test techniques and *mutation* in Figure 4.9, because s_2 is less affected by outliers.

Furthermore, we see that the *W-method* indeed generally performs worse than the *partial W-method*. *Random words* and *random walks* perform similarly well. Figure 4.9 shows that the performance of KV combined with some testing algorithm is similar to the performance of L^* combined with the same testing algorithm. For these reasons and to ease readability, we will ignore certain combinations in the following. In the remainder of this section, we will not show performance plots for combinations involving the *W-method*, random-walks-based testing, or the KV algorithm.

Figure 4.10 shows a cactus plot describing how many learning experiments can reliably be completed with a limited number of test steps. For instance, with RS-*mutation* we are able to learn about 28 models with at most approximately 10,000 test steps, whereas L^*-*mutation* requires about 100,000 test steps to learn only 25 models. We see a steep increase in the required test steps for random-words-based testing to learn three of the 39 models. This explains the discrepancy between s_1-score and and s_2-score of random-words-based testing. It is interesting to note that L^* combinations require a very low number of test steps to learn eight of the models. In general, L^* combinations perform worst, though.

Selected Findings

Next, we discuss a few selected findings.

Counterexample Processing. In Figure 4.8 and Figure 4.9, we see that *mutation* combined with RS and *mutation* combined with TTT perform best overall. In contrast to that, *mutation* combined with KV

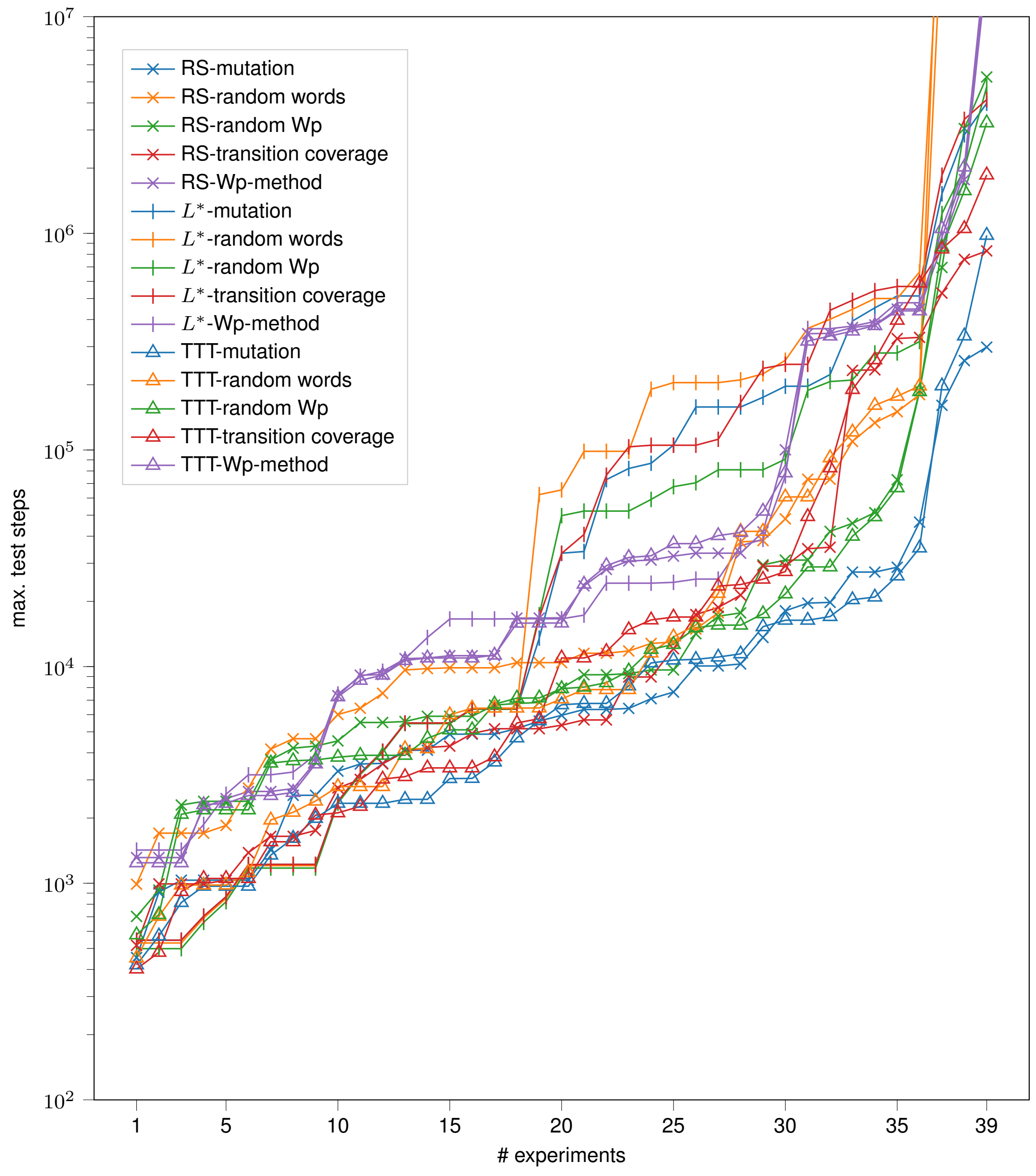


Figure 4.10: A cactus plot showing how many learning experiments can be completed successfully with a limited number of test steps

and *mutation* combined with L^* perform substantially worse, whereas *random Wp* shows uniform performance across different combinations with learning algorithms. Similar observations as for *mutation* can be made for *transition coverage*.

This can be explained by considering the counterexample-processing techniques of different learning algorithms. RS processes counterexamples by extracting distinguishing suffixes [240], like TTT which also performs additional processing steps [151]. This reduces the length and number of sequences that are added to the learning data structures. L^* and KV do not apply such techniques, therefore the performance of these learning algorithms suffers from long counterexamples. We have chosen the parameters for *mutation* conservatively to create long test cases, which leads to long counterexamples, explaining our observations. In contrast to this, *random Wp* generates much shorter test cases. Therefore, we see uniform performance in combination with different learning algorithms. Hence, *mutation* and *transition coverage* should be combined with either RS or TTT. In such combinations, mutation-based testing performs efficient equivalence queries, while sophisticated counterexample processing ensures that a low number of short output queries is performed. Comparing RS and TTT combined with *mutation*,

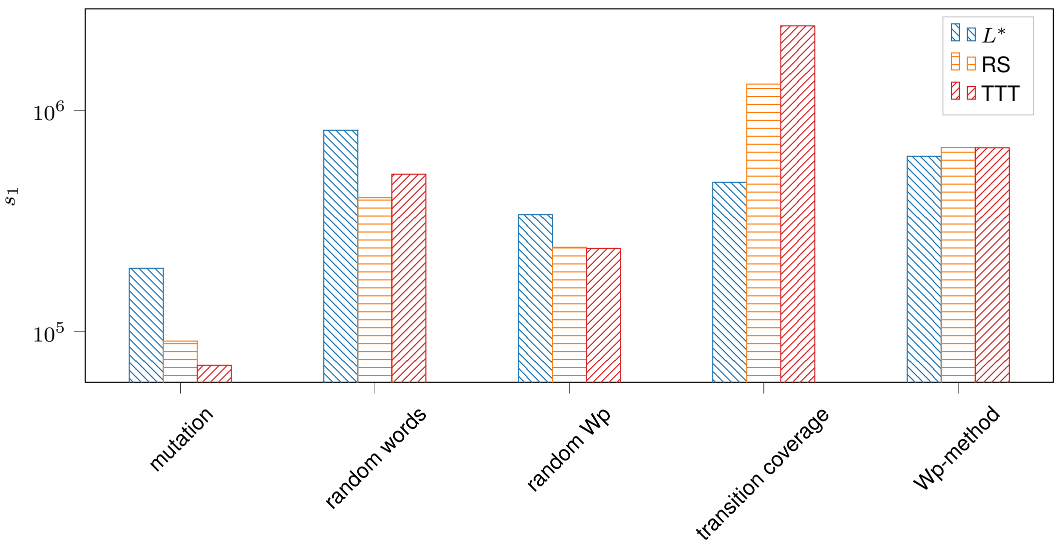


Figure 4.11: The score s_1 computed for experiments involving small models with a sink state

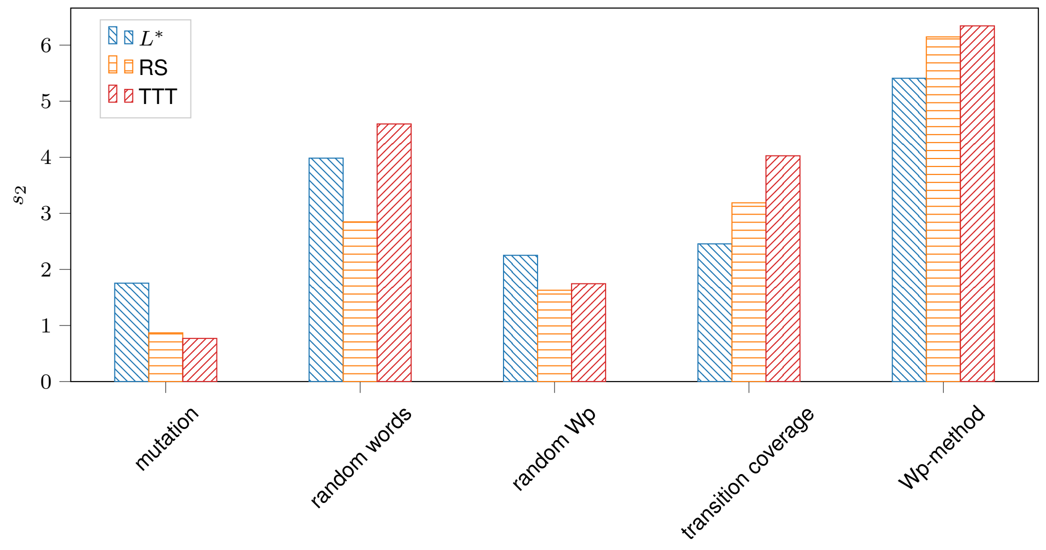


Figure 4.12: The score s_2 computed for experiments involving small models with a sink state

there is no clear winner; both combinations performed similarly well in our experiments.

Small Models with Sink State. We evaluated the learner-tester combinations on ten small models that have a sink state. Small models may result from harsh abstraction. Sink states may be created if learning focuses on individual sessions in a communication protocol, where the sink state is reached upon session termination. Hence, this is an important class of systems that we can identify prior to learning. Therefore, it makes sense to analyse which active automata learning configurations work well in such scenarios.

Figure 4.11 and Figure 4.12 show scores computed for this kind of models. The non-normalised score s_1 shows that transition-coverage-based testing may be very inefficient for such models. In particular, the combinations with RS and TTT are the two worst-performing with respect to s_1. However, the normalised score s_2 is in a similar range as the s_2 score of random-words-based testing. This suggest that the s_1 score is affected by a few experiments for which *transition coverage* performs very poorly. The cactus plot shown in Figure 4.13 demonstrates that this is indeed the case. There is a steep increase in the test steps required to reliably learn in 7 or more experiments. Thus, four models seem to be difficult to learn with *transition coverage*.

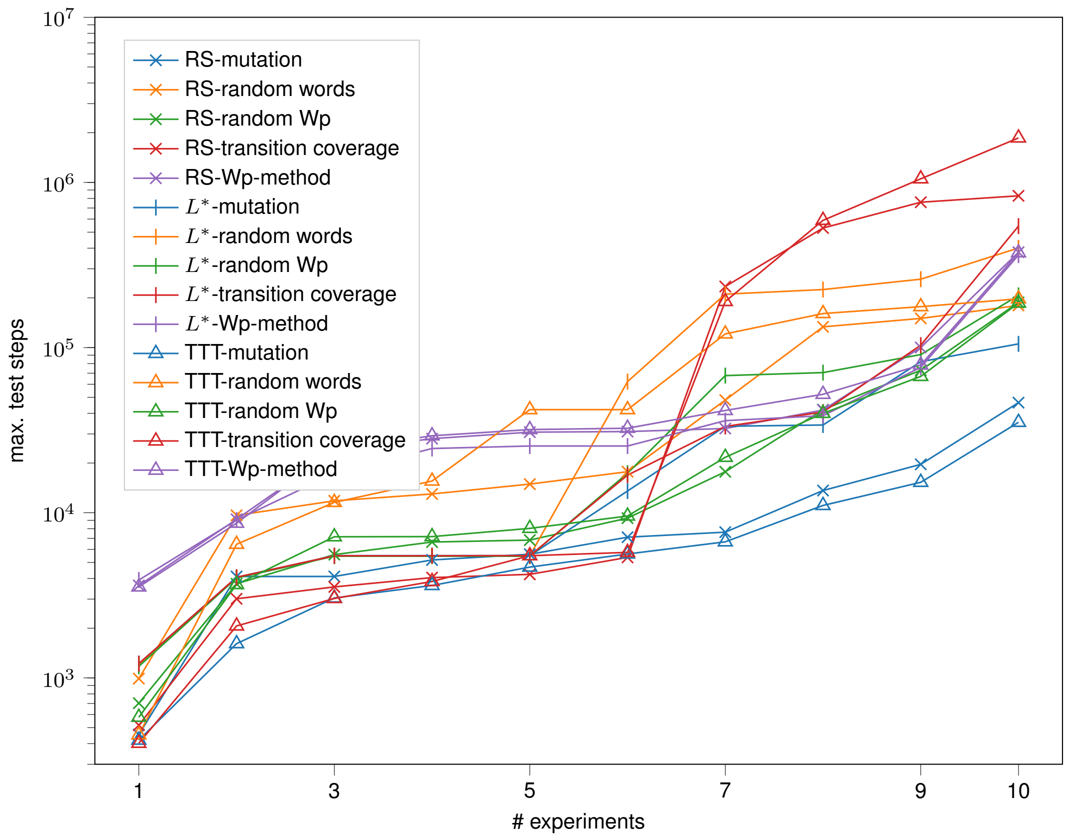


Figure 4.13: A cactus plot showing showing how many learning experiments involving small models with a sink state can be completed successfully with a limited number of test steps

We analysed one of these models in more detail to determine the reason for the poor performance of *transition coverage*. It is a coffee-machine model similar to the model used as an illustrative example by Steffen et al. [258]. Figure 4.14 shows the corresponding Mealy machine. Two properties of the coffee machine cause the poor performance of transition-coverage-based testing. First, many input sequences reach the sink state q_5 that only produces error outputs. Second, other states require very specific input sequences. In experiments, we observed that learning frequently produced incorrect models with 5 states that did not include q_1 or q_2. The transition-coverage heuristic does not help to detect these states. In fact, it is even detrimental. To reach q_1 or q_2, we need to reach the initial state q_0 first. Consequently, covering any known hypothesis transition other than the *water* (*pod*) transition in q_0 leads away from detecting q_2 (q_1). Random testing from q_0 is necessarily more effective. Moreover, the transition-coverage heuristic generates very long test cases. For this reason, most suffixes of these test cases merely execute the self-loop transitions in q_5, because the probability of reaching q_5 is high. This worsens the performance of *transition coverage* even more.

It is interesting to note that mutation-based test-case generation performs well on the coffee machine, although it generates test cases using Algorithm 4.1, like transition coverage. Hence, mutation-based test-case selection is able to drastically improve performance, as can be seen in Figure 4.13. This can be explained by considering the same situation as outlined above. Suppose that we learned an intermediate hypothesis with five states. In this scenario, the true model is a mutant of the hypothesis. By covering that mutant, we detect the last remaining state and learn the true model.

Large Models. Finally, we want to examine the learning performance on large models. In our classification, models are large if they have more than 15 states. Our benchmark set includes 11 such models. Figure 4.15 and Figure 4.16 show scores computed for learning these models. Figure 4.17 shows the corresponding cactus plots.

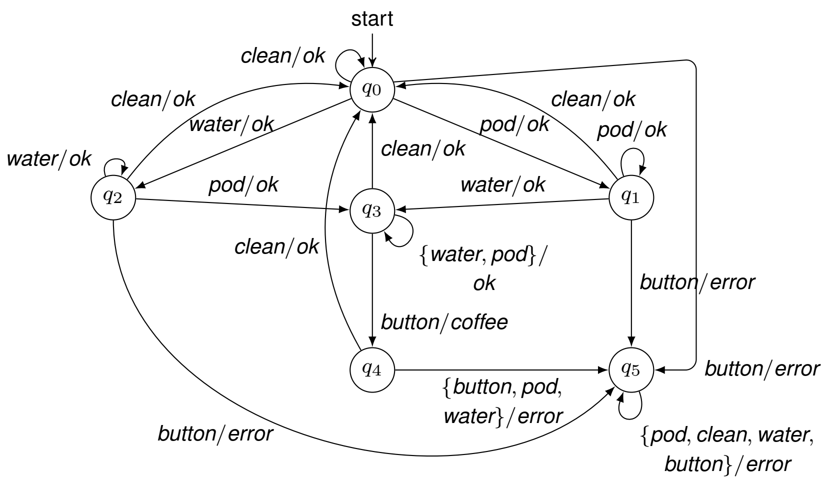


Figure 4.14: A Mealy-machine model of a coffee machine [258]

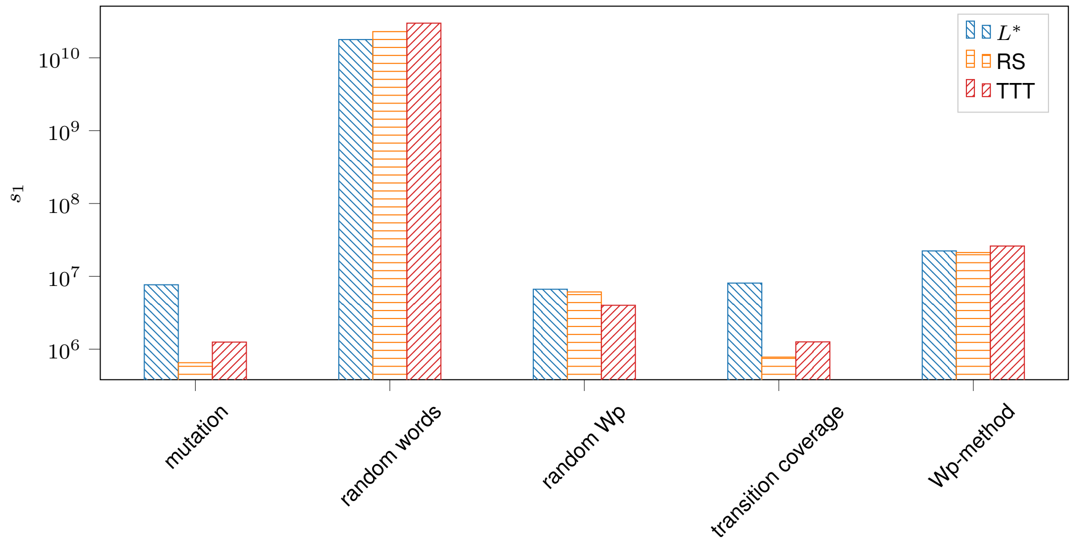


Figure 4.15: The score s_1 computed for experiments involving large models

We can observe that both *random words* and the *Wp-method* show poor performance. Their detailed performance characteristics is different, though. On the one hand, we see in Figure 4.17 that the *Wp-method* combined with any learning algorithm performs bad over the whole range of experiments. On the other hand, *random words* is able to efficiently learn eight of 11 models, but it requires a very large amount of test steps for the remaining three models. These are the TCP-server models, which are much larger than the other considered models. Hence, random-words-based testing is only feasible for moderately large models. We can also observe that the combinations RS-*mutation* and RS-*transition coverage* perform very well for large models. This is in line with findings from Section 4.4.

Discussion of Benchmarking Results

We performed experiments with 28 learner-tester combinations and identified a few representative combinations that we analysed in more detail. As expected, deterministic conformance testing showed poor performance for large models, i.e., it does not scale. Random-words-based testing also cannot reliably learn large models with a limited number of test steps. Transition coverage showed weaknesses for small models with sink states, an important class of system models. Hence, *mutation* or *random Wp* should be

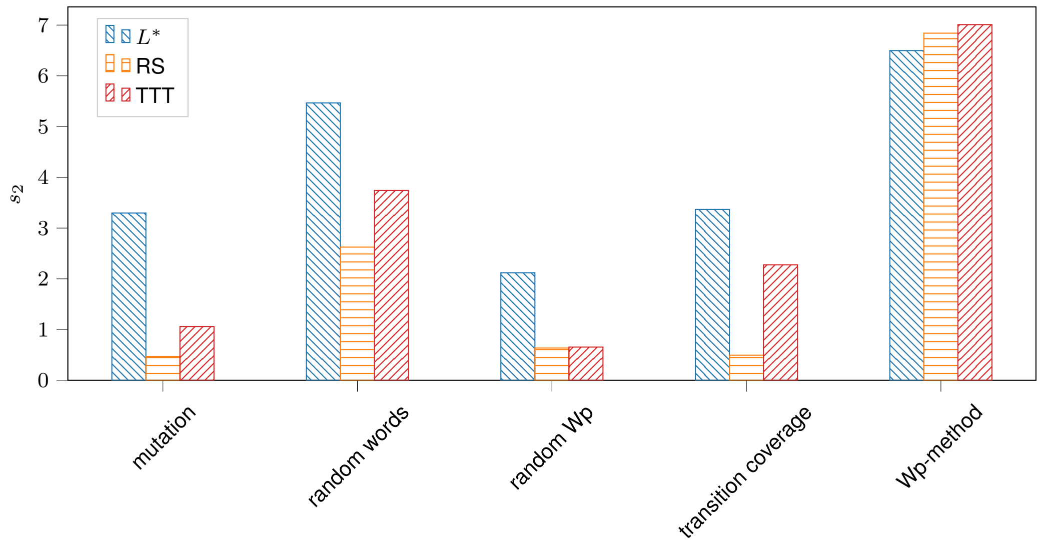


Figure 4.16: The score s_2 computed for experiments involving large models

chosen to efficiently learn automata.

We have also observed that counterexample processing has a large impact on efficiency, especially if test cases are long, as is the case in mutation-based testing. Overall, the combinations RS-*mutation* and TTT-*mutation* performed best. Mutation-based testing requires a low number of test cases for equivalence queries, while sophisticated counterexample processing keeps the number and length of output queries low.

4.6 Summary

This chapter addressed efficiency in active automata learning. We focused on conformance testing for implementing equivalence queries efficiently, because equivalence queries can be considered the main bottleneck of active automata learning with respect to runtime [54]. To tackle this issue, we presented an efficient test-case generation technique which accompanied with appropriate test-case selection yields effective test suites. In particular, we further motivated and described a fault-based test-case selection approach with a fault model tailored towards learning. We performed various experiments in the domain of communication protocols. They showed that it is possible to reliably learn system models with a significantly lower number of test cases than with complete conformance testing, for instance, via the partial W-method [117]. Additionally, we presented an evaluation of combinations of learning algorithms and testing techniques with respect to overall learning performance. This evaluation demonstrated that our fault-based testing approach performs favourably if combined with appropriate learning techniques.

4.7 Results and Findings

We evaluated the performance of our mutation-based test-suite generation in Section 4.4 as well as active automata learning configurations more broadly in Section 4.5. We discussed limitations and potential issues in Section 4.4.5, such as the large number of parameters of randomised testing techniques. In the following, we discuss the evaluation results and our findings concerning the research questions listed in Section 1.6.3. In particular, we address the research questions related to runtime in the work on this chapter.

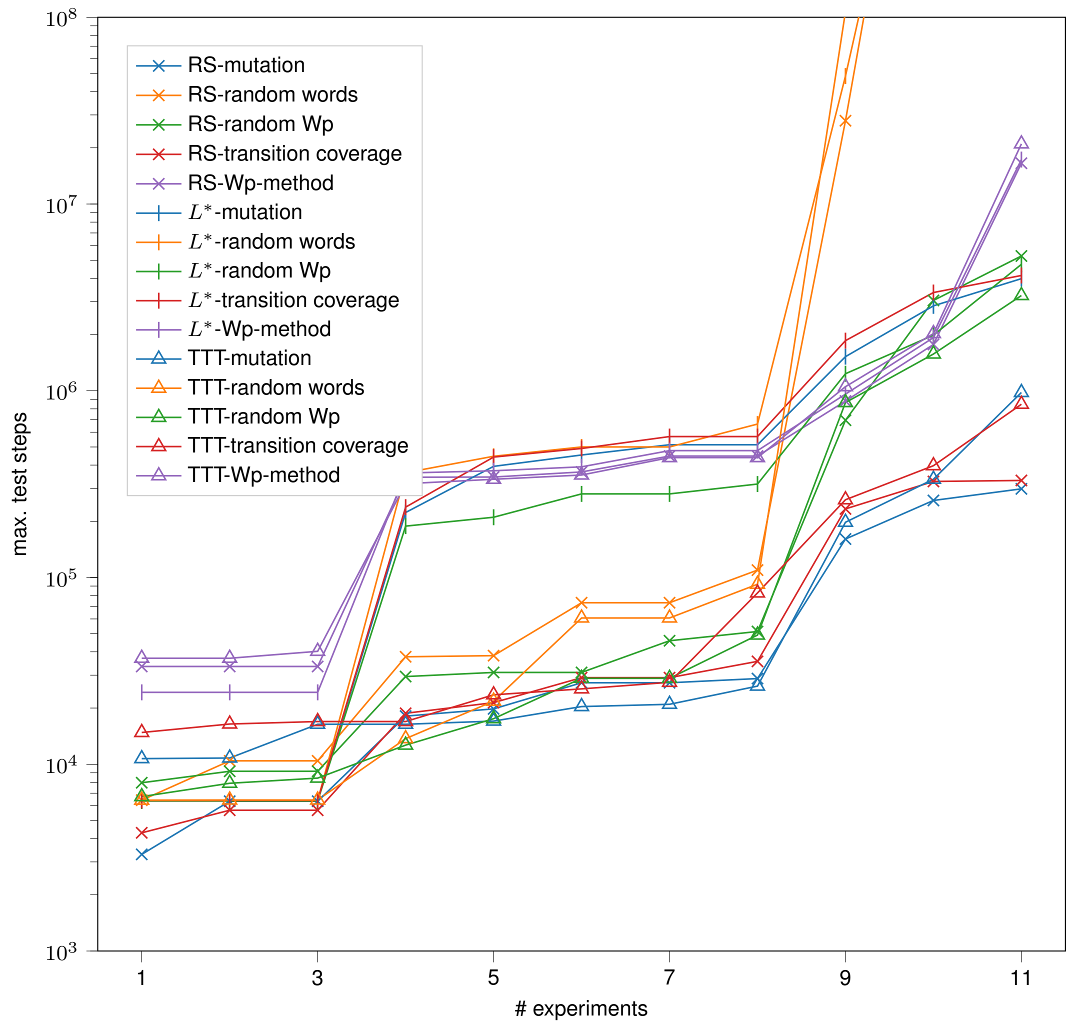


Figure 4.17: A cactus plot showing showing how many learning experiments involving large models can be completed successfully with a limited number of test steps

RQ 1.1 Are randomised testing techniques a sensible choice for learning-based testing? On the one hand, completely random testing did not perform well in our experiments. For instance, the experiments presented in Section 4.5 showed that large models require an enormous amount of completely random tests for reliable learning. On the other hand, the randomised testing techniques that generate test cases from learned models performed very well. Hence, it is essential for testing techniques to exploit knowledge gained throughout learning. *Mutation* and *transition coverage* optimise generated test suites for different forms of coverage of intermediate hypothesis models. As explained in Section 4.1, *random L & Y* and *random Wp* also cover hypothesis states through the selection of test-case prefixes. We found that these four randomised techniques outperformed the deterministic partial W-method significantly.

Hence, we conclude that randomised testing techniques can enable efficient active automata learning and learning-based testing. Such techniques, however, need to exploit knowledge gained during learning to be effective.

RQ 1.2 What guarantees can be given if randomised testing is applied? The goal of the evaluation in Section 4.4 was to empirically demonstrate the effectiveness of our mutation-based test-suite generation, where we also briefly discussed guarantees in this chapter.

First, we can provide similar guarantees as model-based mutation testing in general. By executing a test case covering a mutant, we can show that this particular mutant has not been implemented. Second, we benefit from the minimality of actively learned automata models. If we learned a model with n states, then we know that we either learned the true model, or that the true model has more than n states. As explained in Section 4.4.5, this is similar to the guarantees provided by deterministic conformance testing using the partial W-method [117]. In fact, conditioned on limited test-execution time, learning supported by randomised testing may be able to provide stronger guarantees than learning supported by deterministic testing.

RQ 1.3 Can learning with randomised conformance testing reliably generate correct system models? We conclude from our experiments presented in Section 4.4 and Section 4.5 that learning supported by randomised conformance testing can reliably generate correct system models. In particular, the mutation-based testing approach performed well. It was able to reliably learn models of all systems considered in Section 4.4 and it achieved the best scores in Section 4.5, if combined with the RS algorithm or the TTT algorithm.

RQ 1.4 Is fault-based testing, such as model-based mutation testing, applicable in automata learning? We developed and successfully applied a fault-based testing technique, therefore we conclude that a fault-based approach, implemented via model-based mutation testing, is applicable in automata learning.

It should be noted, though, that not all types of mutations are equally effective. As mentioned in Section 4.2.3, the *transition coverage* testing technique can be implemented by *change output* mutations. This testing strategy showed poor performance in some cases. We observed poor performace for small models with sink states, as pointed out in Section 4.5. Actually, *change output* mutations do not model realistic faults in the context of active automata learning. Mutants created by solely changing outputs cannot represent the true SUL, due to the minimality of learned hypotheses. A learned hypothesis either has strictly less states than the true model of the SUL or it is correct and minimal, that is, every transition including its output is correct.

Concluding Remarks. In summary, we conclude that randomised mutation-based test-suite generation is a valuable technique for conformance testing in active automata learning. In general, coverage-guided randomised testing can be considered promising in this area, as it neatly combines exploitation of gained knowledge in the form of hypotheses with exploration through randomisation. This form of testing also requires little prior knowledge about SULs which suits our black-box view in automata learning.

5 Modelling and Learning of Uncertain Behaviour

Declaration of Sources

This chapter introduces background information on stochastic models, verification, and learning. Our work on stochastic automata learning is covered by four publications [16, 18, 265, 266], therefore we base the presentation of material in this chapter mostly on these publications.
The background information covers (1) basics, like probability distributions and corresponding notations, (2) Markov decision processes (MDPs), which is the modelling formalism we used, (3) automata learning, (4) probabilistic model-checking and reachability objectives, and (5) statistical model-checking (SMC).

5.1 Choice of Modelling Formalism for Learning-Based Testing of Uncertain Behaviour

In our exploratory research, we identified uncertain behaviour as an issue limiting the applicability of deterministic learning-based testing; see Section 1.6.4 and Section 3.6. Recall that we consider behaviour to be uncertain if the repeated application of the same input sequence results in different output sequences. This kind of behaviour renders the application of deterministic automata learning such as the L^* algorithm [37] infeasible. In order to perform learning-based testing in the presence of uncertainties, we need to apply a learning technique that can cope with uncertainties. **RQ 2.1** addresses the question of which modelling formalism and, consequently, which learning technique should be used in this context. To approach this question, we present examples from the literature on learning uncertain behaviour in the following.

Most L^*-based work targets deterministic models, but there are exceptions to that. We want to discuss two approaches to learning non-deterministic models that consider a testing scenario [161, 283]. Volpato and Tretmans presented an adaptation of Angluin's L^* for learning non-deterministic input-output transition systems [283]. They note that learned models can be applied in model-based testing and that testing can be used in their learning algorithm. However, they discuss learning only abstractly on the basis of an oracle (a teacher). In particular, they assume an oracle capable of answering output queries. Such output queries non-deterministically return an output that can be produced after a given system trace. Hence, these queries need to be called repeatedly to determine all possible outputs. In addition to output queries, they assume that the oracle can tell, when all outputs after a given trace have been observed. Khalili and Tacchela presented an active learning algorithm for non-deterministic Mealy machines [161] which extends concepts introduced by Angluin [37] as well. They also note that they need to perform tests repeatedly to observe all possible outputs. Unlike Volpato and Tretmans, they discuss practical issues. They assume that there are no rare events to ensure that all possible outputs can be observed in an acceptable amount of time. Their implementation of output queries is parameterised by a minimum probability of observing some output and by a confidence value. Thus, they consider an implicit stochastic behaviour model, but they abstract away from observed output frequencies and learn purely non-deterministic models. What is more, they actually suggest that a stochastic (MDP-based) interpretation of learned models can be used for effective testing during equivalence queries.

We gained an important insight from studying these approaches. A practical implementation of a learning technique for non-deterministic models requires some form of stochastic interpretation of the SUL [161]. This is necessary to be able to tell when all non-deterministically produced outputs have likely been observed. Therefore, we decided to study modelling formalisms and learning techniques for stochastic systems. We found MDPs to be well-suited models for our application domain of networked systems for the following reasons. MDPs are commonly used to model randomised distributed algorithms [46] and more specifically to model network protocols for verification [173, 220]. This kind of models can be controlled via inputs like Mealy machines, while transitions are stochastic, thus enabling testing. There exist learning algorithms for MDPs as well. IoAlergia is a state-of-the-art passive learning algorithm for MDPs that has already successfully been applied in a verification context [198, 199].

Another benefit of MDPs is that they potentially encode more information than non-deterministic models like non-deterministic Mealy machines. They model the probability of events which undoubtedly matters in practical scenarios. Consequently, **we decided to model uncertain behaviour as stochastic behaviour by choosing MDPs as modelling formalism**. As a first step, we developed a learning-based testing approach relying on IoAlergia [198, 199] for learning, as we will discuss in Chapter 6.

5.2 Basics

As in our previous publications on stochastic automata learning [16, 18, 265, 266], we introduce background material mostly following Mao et al. [199] and Forejt et al. [114]. An important difference to these works [114, 199] is that we mainly consider finite traces and finite paths, as we follow simulation-based approaches that generate traces via testing. In contrast to that, model checking mostly considers properties defined over infinite sequences [114]. We also use slightly different notation.

5.2.1 Probability Distributions

Given a set S, we denote by $Dist(S)$ the set of probability distributions over S, thus for all μ in $Dist(S)$ we have $\mu : S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. In this thesis, distributions μ may be partial functions, in which case we implicitly set $\mu(e) = 0$ if μ is not defined for e. For $A \subseteq S$, $\mathbf{1}_A$ denotes the indicator function of A, i.e. $\mathbf{1}_A(e) = 1$ if $e \in A$ and $\mathbf{1}_A(e) = 0$ otherwise. Hence, $\mathbf{1}_{\{e\}}$ for $e \in S$ is the probability distribution assigning probability 1 to e.

We apply the two pseudo-random functions *coinFlip* and *rSel* in Chapter 6 and in Chapter 7. Section 1.9 introduces these auxiliary functions.

5.2.2 String Notation

Let I and O be sets of input and output symbols. As explained below, outputs label states and inputs label edges in MDPs, like in Moore machines. Hence, traces of MDPs are usually alternating sequences of outputs and inputs that start and end with an output, because paths corresponding to traces that start and end in a state. A trace is therefore an input/output string s which is an alternating sequence of inputs and outputs, starting with an output, i.e. $s \in O \times (I \times O)^*$. We extend the general notational conventions for sequences introduced in Section 1.9, like concatenation and indexed access, to such input/output string. The first element in such a string is generally the first input-output pair and we access the initial output explicitly, if required. Furthermore, the length of an input/output string is its number of pairs, thus traces consisting of only an initial output have length zero. Prefix- and suffix-closedness are adapted analogously. In slight abuse of notation, we use $A \times B$ and $A \cdot B$ interchangeably to simplify notation.

5.3 Markov Decision Processes

MDPs allow modelling reactive systems with probabilistic responses. An MDP starts in an initial state. During execution, the environment chooses and executes inputs non-deterministically upon which the system reacts according to its current state and its probabilistic transition function. For that, the system changes its state and produces an output.

Definition 5.1 (Markov decision process (MDP)).
A labelled Markov decision process (MDP) is a tuple $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$ where

- *Q is a finite set of states,*
- *I is a finite set of input symbols,*
- *O is a finite set of output symbols,*
- *$q_0 \in Q$ is the initial state,*
- *$\delta : Q \times I \to Dist(Q)$ is the probabilistic transition function, and*
- *$L : Q \to O$ is the labelling function.*

The transition function δ must be defined for all $q \in Q$ and $i \in I$, thus MDPs are input enabled in our definition. We consider only deterministic MDPs, therefore it must hold that

$$\forall q, q', q'' \in Q, \forall i : (\delta(q,i)(q') > 0 \wedge \delta(q,i)(q'') > 0) \to (q' = q'' \vee L(q') \neq L(q'')).$$

Non-determinism results only from the non-deterministic choice of inputs by the environment.

We generally consider deterministic labelled MDPs. Labelling of states with outputs allows us to distinguish states in a black-box setting, thus it is essential for us. Deterministic MDPs define at most one successor state for each source state and input-output pair, which ensures that a given trace always reaches the same state (see also below). This assumption simplifies learning such that learning algorithms, like IoAlergia [199], generally place this assumption on SULs. In this thesis, we refer to deterministic labelled MDPs uniformly as MDPs.

As a shorthand notation, we use $\Delta : Q \times I \times O \to Q \cup \{\bot\}$ to compute successor states for a given source state and an input-output pair. The function is defined by $\Delta(q, i, o) = q' \in Q$ with $L(q') = o$ and $\delta(q, i)(q') > 0$ if there exists such a q', otherwise Δ returns $\bot$.

In the context of verification, labels are often propositions that hold in states [114], thus $O = \mathcal{P}(AP)$, where AP is a set of relevant (atomic) propositions, and $L(q)$ returns the propositions that hold in state q. Additionally to requiring determinism and labelling, Definition 5.1 requires MDPs to be input enabled. They must not block or reject inputs. Since we assume stochastic SUTs to be MDPs, this allows us to execute any input at any point in time. This is a common assumption in model-based testing [271].

Example 5.1 (Faulty Coffee Machine). Figure 5.1 shows an MDP modelling a faulty coffee machine. Edge labels denote input symbols and corresponding transition probabilities, whereas output labels in curly braces are placed above states. After providing the inputs `coin` and `but`, the coffee machine MDP produces the output `coffee` with probability 0.9, but with probability 0.1 it resets itself producing the output `init`.

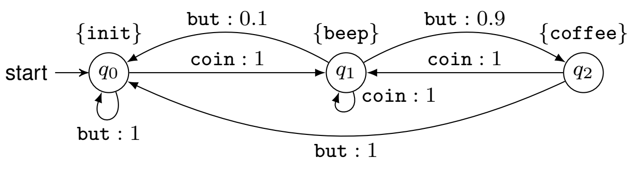


Figure 5.1: An MDP modelling a faulty coffee machine

5.3.1 Execution of Markov Decision Processes

A (finite) path ρ through an MDP is an alternating sequence of states and inputs starting in the initial state and ending in some state $q_n \in Q$, that is, $\rho = q_0 \cdot i_1 \cdot q_1 \cdots i_{n-1} \cdot q_{n-1} \cdot i_n q_n \in Q \times (I \times Q)^*$. The set of all paths of an MDP $\mathcal{M}$ is denoted by $Path_{\mathcal{M}}$. In each state q_k, the next input i_{k+1} is chosen non-deterministically and based on that, the next state q_{k+1} is chosen probabilistically according to $\delta(q_k, i_{k+1})$. Hence, we have for each k that $\delta(q_k, i_{k+1}) > 0$. In contrast to finite paths, infinite paths do not have a dedicated end state. An infinite path $\hat{\rho}$ is sequence $q_0 \cdot i_1 \cdot q_1 \cdot i_2 \cdots$ [114]. We denote the set of infinite paths of $\mathcal{M}$ by $IPath_{\mathcal{M}}$. Unless otherwise noted, we refer to finite paths simply as paths, as we generally consider a test-based setting in which we execute finite paths.

The execution of an MDP is controlled by a so-called scheduler, resolving the non-deterministic choice of inputs. A scheduler, as defined below, specifies a distribution over the next input given the current execution path. In other words, they basically choose the next input action (probabilistically) given a history of visited states. Schedulers are also referred to as adversaries or strategies [199].

Definition 5.2 (Scheduler).
Given an MDP $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$, a scheduler for $\mathcal{M}$ is a function $s : Path_{\mathcal{M}} \to Dist(I)$.

The composition of an MDP $\mathcal{M}$ and a scheduler s induces a deterministic Markov chain [114]. A Markov chain is a fully probabilistic system allowing to define a probability measure over paths. Below, we define a probability measure over finite paths based on the definition in our article on learning-based testing [18].

Probability Distributions on Paths. For a probability distribution over finite paths of an MDP $\mathcal{M}$ controlled by scheduler s, we also need a probability distribution $p_l \in Dist(\mathbb{N}_0)$ over the path lengths.

Definition 5.3 (Path Probabilities).
An MDP $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$, a scheduler $s : Path_{\mathcal{M}} \to Dist(I)$, and a path length probability distribution p_l induce a probability distribution $\mathbb{P}^l_{\mathcal{M},s}$ on finite paths $Path_{\mathcal{M}}$ defined by:

$$\mathbb{P}^l_{\mathcal{M},s}(q_0 \cdot i_1 \cdot q_1 \cdots i_n \cdot q_n) = p_l(n) \cdot \left(\prod_{j=1}^{n} s(q_0 \cdots i_{j-1} q_{j-1})(i_j) \cdot \delta(q_{j-1}, i_j)(q_j) \right) \tag{5.1}$$

Probability distributions over finite paths may instead of p_l, for instance, include state-dependent termination probabilities [197]. We take a path-based view because we actively sample from MDPs. Moreover, probability distributions are often defined for each state and are therefore parameterised by states [114]. Since we sample all SUL traces starting from the initial state q_0, Equation (5.1) defines only probabilities of paths starting in q_0.

Verification techniques, such as model checking, usually reason about infinite paths [46, 114]. Consequently, probability spaces are defined over infinite paths, where the basis for the construction of such probability spaces is the probability of executing a finite prefix of a set of infinite paths. Since we take a test-based view, we consider only finite paths. For more information, we refer to Forejt et al. [114] and Baier and Katoen [46].

Scheduler Subclasses. There are subclasses of schedulers that are relevant to us. Since we mainly target reachability in Chapter 6, we do not need general schedulers, but may restrict ourselves to *memoryless deterministic* schedulers [172]. A scheduler is memoryless if its choice of inputs depends only on the current state, thus it is a function from the states Q to $Dist(I)$. A scheduler s is deterministic if for all $\rho \in Path_{\mathcal{M}}$, there is exactly one $i \in I$ such that $s(\rho)(i) = 1$. Otherwise, it is called randomised. Example 5.2 describes an MDP and a scheduler for our faulty coffee machine introduced in Example 5.1.

Note that bounded reachability, as considered in Chapter 6, actually requires finite-memory schedulers. However, bounded reachability can be encoded as unbounded reachability by transforming the MDP model [69], at the expense of increased state space.

Example 5.2 (Scheduler for Coffee Machine). A deterministic memoryless scheduler s may provide the inputs `coin` and `but` in alternation to the coffee machine of Example 5.1. Formally, $s(q_0) = 1_{\{\texttt{coin}\}}$, $s(q_1) = 1_{\{\texttt{but}\}}$, and $s(q_2) = 1_{\{\texttt{coin}\}}$. By setting the length probability distribution to $p_l = \mathbf{1}_{\{2\}}$, all strings must have length 2, such that, for instance, $\mathbb{P}^l_{\mathcal{M},s}(\rho) = 0.9$ for $\rho = q_0 \cdot \texttt{coin} \cdot q_1 \cdot \texttt{but} \cdot q_2$.

Traces. During the execution of a finite path ρ, we observe a trace $L(\rho) = t$. As mentioned above, a trace is an alternating sequence of inputs and outputs starting with an output. The trace t observed during the execution of $\rho = q_0 \cdot i_1 \cdot q_1 \cdots i_{n-1} \cdot q_{n-1} \cdot i_n \cdot q_n$ is given by $t = o_0 \cdot i_1 \cdot o_1 \cdots i_{n-1} \cdot o_{n-1} \cdot i_n \cdot o_n$ where $L(q_i) = o_i$.

Since we consider deterministic MDPs, L is invertible, thus each trace in $O \times (I \times O)^*$ corresponds to at most one path and $\mathbb{P}^l_{\mathcal{M},s}$ can be adapted to traces t by defining:

$$\mathbb{P}^l_{\mathcal{M},s}(t) = \begin{cases} \mathbb{P}^l_{\mathcal{M},s}(\rho) & \ldots \text{ if there is a } \rho \text{ with } L(\rho) = t \\ 0 & \ldots \text{ otherwise} \end{cases}$$

By controlling path length via p_l, we also control the length of observed traces via p_l.

5.4 Learning Stochastic Automata

An influential learning algorithm for stochastic automata is ALERGIA [74]. ALERGIA learns stochastic finite automata which are essentially discrete-time Markov chains with state-dependent termination probabilities. Each state in such an automaton specifies a probability stating that the produced sequence ends in that state. ALERGIA has been adapted to learn MDPs [198, 199]. This adaptation is called IOALERGIA, as learned automata distinguish between inputs and outputs. IOALERGIA takes a multiset of traces as input and as a first step, it constructs an input output frequency prefix tree acceptor (IOFPTA) representing the traces. An IOFPTA is a tree with edges labelled by inputs and nodes labelled by outputs. It concisely represents the input sample of traces. Additionally, edges are annotated with frequencies denoting how many traces corresponding to an edge are included in the input sample. An IOFPTA with normalised frequencies represents a tree-shaped MDP where tree nodes correspond to MDP states.

In a second step, the IOFPTA is transformed through iterated state-merging, which potentially introduces cycles. This step compares nodes in the tree and merges them if they show similar output behaviour such that it is likely that they correspond to the same state of the SUL. IOALERGIA basically views the IOFPTA as an MDP with non-normalised transition probabilities.

More concretely, the state-merging algorithm IOALERGIA initialises the learned non-normalised MDP with the IOFPTA. During the operation of this algorithm, the learned MDP is continuously updated. The states of this MDP are partitioned into three sets: *red* states which have been checked and which are states of final learned MDP, *blue* states which are neighbours of red states, and uncoloured states. Initially, the only red state is the root of the IOFPTA. After initialisation, pairs of blue and red states are checked for compatibility and merged if compatible. If there is no compatible red state for a

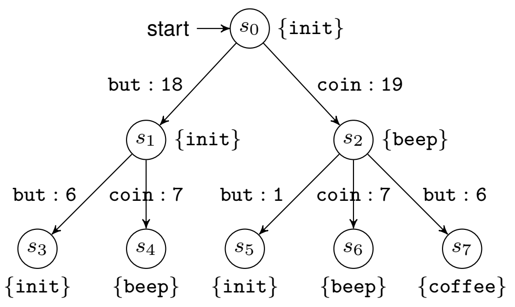


Figure 5.2: An IOFPTA of the faulty coffee machine

blue state, the blue state is coloured red. This is repeated until all compatible states have been merged. After normalisation of transition probabilities, IOALERGIA returns the final learned MDP.

Roughly speaking, two states are compatible if they have the same label, their outgoing transitions are compatible and their successors are recursively compatible [199]. Outgoing transitions are compatible, if their empirical probabilities, estimated from the input data, are sufficiently close to each other. In other words, we check for all inputs if the estimated probability distribution over outputs conditioned on inputs are statistically similar. If they are, we check recursive compatibility of successors reached by all input-output pairs. In IOALERGIA, a parameter $\epsilon_{\text{ALERGIA}} \in (0, 2]$ controls the significance level of a statistical test, which determines whether two empirical probabilities are sufficiently close to each other [199]. We represent calls to IOALERGIA by IOALERGIA$(\mathcal{S}, \epsilon_{\text{ALERGIA}}) = \mathcal{M}$ where $\mathcal{S}$ is a multiset of traces and $\mathcal{M}$ is the learned MDP.

Example 5.3 (IOFPTA of Coffee Machine). Figure 5.2 shows an IOFPTA for the coffee machine from Example 5.2, computed from traces sampled with a (uniformly) randomised scheduler and a p_l with a support that includes $\{0, 1, 2\}$. Edge labels denote inputs and associated frequencies, while outputs are placed next to nodes. In the first step of state-merging, s_0 would be coloured red and its neighbours, s_1 and s_2, would be coloured blue. Then, s_1 might be chosen first to be checked for compatibility with s_0. As the successors of s_0 and s_1 are similar, s_1 would be merged with s_0 (depending on $\epsilon_{\text{ALERGIA}}$). Redirecting the `but` edge from s_1 to s_0 would create the self-loop transition in the initial state.

5.5 Property Specification

There exist various specification languages for probabilistic model-checking, including probabilistic computation tree logic (PCTL) [46, 114] which is also supported by the probabilistic model-checker PRISM [174]. Generally, PCTL distinguishes between *state* formulas and *path* formulas. State formulas include a probabilistic path operator $\mathbf{P}_{\oplus p}[\phi]$, where ϕ is a path formula. This operator is satisfied in a state q, if for all schedulers, the probability p' of taking an (infinite) path in q which satisfies ϕ is such that $p' \oplus p$, where $\oplus \in \{\leq, <, >, \geq\}$ and $p \in [0, 1]$. Moreover, state formulas may be combined through Boolean operators. Path formulas are formed using temporal operators, such as the *next* and the *until* operator. A path formula ϕ basically defines a set of paths that satisfy ϕ. PRISM also supports an extension of the probabilistic path operator, the operator $\mathbf{P}_{\max=?}[\phi]$, which computes the maximum probability of satisfying ϕ quantified over all schedulers.

Since we use only a small subset of PCTL, we refer to [46, 114] for the precise syntax and semantics of PCTL. In our evaluation of test-based learning of MDPs in Section 7.6, we provide a description of all PCTL properties in natural language. In learning-based testing of stochastic systems in Chapter 6, we use a special form of properties that we describe below. We check the maximum probability of step-bounded reachability. Step-bounded reachability can be derived from the *bounded until* operator [174].

5.5.1 Step-Bounded Reachability

The syntax of the step-bounded reachability formulas ϕ that we consider in Chapter 6 is given by:

$\phi = F^{<k}\psi$ with $\psi = \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid AP$, AP denoting an atomic proposition, and $k \in \mathbb{N}$.

The formula $\phi = F^{<k}\psi$ denotes that ψ needs to be satisfied in a state reached in less than k steps from the initial state, where ψ is a property defined over output labels. A finite trace $t = o_0 i_1 o_1 \cdots i_{n-1} o_{n-1} i_n o_n$ satisfies $\phi = F^{<k}\psi$, denoted by $t \models \phi$, if there is an $i < k$ such that $o_i \models \psi$. The evaluation of a trace t with respect to a formula $\phi = F^{<k}\psi$ places restrictions on the length of t. In particular, we can only conclude that $t \not\models \phi$ if t does not contain an o_i with $o_i \models \psi$ and contains at least $k-1$ steps. In other words, t must be long enough to determine that it does not satisfy a reachability property. To ascertain that all traces can be evaluated, we set the length probability p_l on paths/traces accordingly when executing MDPs. We generally ensure for all traces that $p_l(j) = 0$ for $j < k-1$, when we sample traces for step-bounded reachability. Note that the length distribution of finite traces is completely specified by p_l as MDPs cannot reach deadlock states, because they are input-enabled by Definition 5.1.

We aim to maximise the probability of satisfying a step-bounded reachability property $F^{<k}\psi$ in Chapter 6. This can be expressed by the formula $\mathbf{P}_{\max=?}[F^{<k}\psi]$ for probabilistic model-checking with PRISM [114].

5.6 Statistical Model-Checking

Statistical model-checking (SMC) provides simulation-based techniques to statistically analyse stochastic systems with respect to properties that are, for instance, expressed in temporal logics [177]. Basically, SMC allows to answer two types of questions: (1) qualitative questions asking whether the probability of satisfying a certain property is above or equal to a given threshold; (2) quantitative questions asking for the probability of satisfying certain properties, relative to a given precision of the probability. In general, SMC simulates a system finitely many times. It then checks whether the traces produced by the simulation satisfy the property under consideration and applies statistical techniques to answer the two types of questions, depending on the simulation outcomes. It is usually more light-weight than probabilistic model-checking, as it does not suffer from state-space explosion. In this thesis, we apply SMC only for quantitative questions asking for the probability of satisfying step-bounded reachability properties.

The composition of a scheduler s and an MDP $\mathcal{M}$ behaves fully probabilistically. It is not affected by non-determinism. In fact, such a composition induces a discrete time Markov chain (DTMC) [114]. As a result, we can apply techniques from SMC without considering non-determinism.

In order to analyse step-bounded reachability properties ϕ, we can define the probability of satisfying ϕ with an MDP $\mathcal{M}$, and a scheduler s by $\mathbb{P}_{\mathcal{M},s}(\phi) = \mathbb{P}^l_{\mathcal{M},s}(\{\rho \in Path_{\mathcal{M}} \mid L(\rho) \models \phi\})$ for an appropriate p_l. Note that the value $\mathbb{P}_{\mathcal{M},s}(\phi)$ does not depend on the actual p_l as long as p_l ensures that traces are long enough to allow reasoning about the satisfaction of ϕ. Put differently, p_l must be such that traces shorter than the step bound specified by ϕ have zero probability.

In order to estimate the probability $\mathbb{P}_{\mathcal{M},s}(\phi)$ via SMC [177], we associate Bernoulli random variables B_i with success probability p with simulations of the MDP $\mathcal{M}$, serving as our SUT, controlled by s. A realisation b_i is 1, if the corresponding sampled trace satisfies ϕ and 0 otherwise. To estimate $p = \mathbb{P}_{\mathcal{M},s}(\phi)$, we apply a Monte Carlo simulation. Given n individual simulations, the estimate $\hat{p}$ is the observed relative success frequency, computed by $\hat{p} = \sum_{i=1}^{n} \frac{b_i}{n}$. In order to bound the error of the estimation with a certain degree of confidence, we compute the number of required simulations based on a Chernoff bound [177, 223]. This bound guarantees that if p is the true probability, then the distance between $\hat{p}$ and p is greater than or equal to some ϵ with a probability of at most δ. Formally it holds that $\mathbb{P}(|\hat{p} - p| \geq \epsilon) \leq \delta$ if we perform sufficiently many simulations. The required number of simulations n and the parameters ϵ and δ are related by $\delta = 2e^{-2n\epsilon^2}$ [223], hence we compute n by

$$n = \left\lceil \frac{\ln(2) - \ln(\delta)}{2\epsilon^2} \right\rceil. \tag{5.2}$$

We commonly refer to ϵ as error bound and to $1 - \delta$ as confidence.

5.7 Discussion

At the beginning of this chapter, we discussed our choice of modelling formalism to represent uncertain behaviour. To conclude this chapter, we shall revisit **RQ 2.1** that addresses this choice.

RQ 2.1 Which modelling formalisms are appropriate in learning-based testing of uncertain behaviour? Models like non-deterministic Mealy machines are a suitable choice for active automata learning via testing, as demonstrated by Khalili and Tacchela [161]. They also showed that a stochastic interpretation of systems facilitates learning. This suggests that stochastic models like MDPs are a more natural choice. Moreover, MDPs model probabilities of events which is not possible with non-deterministic Mealy machines. Due to that, stochastic model learning can utilise more information provided by sampled traces. It can make use of observed frequencies of events and encode them in learned models. Finally, MDPs are often used in verification to model (communication) protocols [47, 78, 102, 105, 173, 220, 242, 295], which are the focus of this thesis. For these reasons, we decided to model uncertain behaviour with MDPs.

It should be noted, though, that non-deterministic models are often used in model-based testing and conformance relations like **ioco** enable testing based on such models [269]. However, non-determinism in test models often serves a specific purpose. It allows for implementation freedom in certain aspects. Since we want to accurately capture the behaviour of implementations through learning, this type of non-determinism is generally not relevant to us.

6 Learning-Based Testing of Stochastic Systems

Declaration of Sources

This chapter discusses a learning-based approach to test stochastic systems with respect to reachability properties. We presented this approach first at the RV 2017 [16]. Following the conference presentation, we extended the approach and its evaluation. We wrote a journal article on our extended work that has been published as part of the special issue on the RV 2017 of the journal *Formal Methods in System Design* [18]. The presentation of the testing approach and its evaluation is mainly based on the extended article.

6.1 Probabilistic Black-Box Reachability Checking

Model checking has a long-standing tradition in software verification. Given a system design, model-checking techniques determine whether requirements stated as formal properties are satisfied. These techniques and other forms of model-based verification fall short if no design is available.

Black Box Checking. Peled et al. [230] presented black box checking as a solution to this problem. Black box checking comprises strategies to check formal properties of black-box systems without a known design. One of these strategies is a technique that combines automata learning, model checking and testing.

This learning-based technique learns models of black-box SUTs in the form of DFA on-the-fly and iteratively via Angluin's L^* [37]. Whenever a hypothesis automaton model is created, the hypothesis is model checked with respect to some property. If model checking determines that the hypothesis does not satisfy the checked property, it returns a counterexample sequence c as a witness. Such a counterexample c is tested on the SUT. This may reveal an actual fault in the SUT causing the process to be stopped as it has successfully found a fault. If testing c does not reveal a fault, the hypothesis and the SUT disagree on the input sequence c. Hence, c is a counterexample to equivalence between hypothesis and SUT. Black box checking uses such a counterexample, comparable to a counterexample returned from an equivalence query, to improve the hypothesis. If model checking determines that the hypothesis satisfies the checked property, black box checking tests for equivalence between SUT and hypothesis via the W-method [83, 277]. In other words, it implements an equivalence query via conformance testing as discussed in Section 2.2.2. In case non-equivalence is detected by testing, the learned hypothesis is improved using a failed test and learning via L^* is resumed. This process continues until either a fault has been detected, or SUT and hypothesis are determined to be equivalent using the W-method. The latter is the usual stopping criterion in active automata learning.

Black box checking enables the verification of black-box systems. However, it has a few shortcomings. It assumes a known upper bound on the number of SUT states. With that assumption, the W-method [83, 277] can be used to implement a perfect equivalence query. Its application is still expensive as explained in Section 2.4.2 and as demonstrated in Chapter 4. What is more, **black box checking** assumes the SUT to be deterministic, thus it **cannot be applied if the SUT shows uncertain behaviour**.

In this chapter, we tackle this limitation to enable black box checking of uncertain behaviour, also by combining model learning, model checking and testing. However, the exact techniques and their combinations are different, as we, for instance, cannot apply Angluin's L^* [37].

Goal and Setting. We consider a testing scenario, in which we know the interface of a black-box system and we can gain information by testing the system. Furthermore, we assume that inputs to the system can be freely chosen and that reactions are stochastic. As noted in Section 5.1, this makes MDPs a well-suited choice of model type. Given such a system, we aim at generating testing strategies that produce desired outputs with high probability. Put differently, we aim at maximising the probability of reaching desired outputs. We call the proposed learning-based testing approach **probabilistic black-box reachability checking**.

We apply IoAlergia [198, 199] for learning. Although this learning technique is passive in general, our technique is active, as we generate new data for learning by testing. In an iterative approach, we steer the data generation based on learned models towards desired outputs to explore relevant parts of the system more thoroughly. That way, we aim at iteratively improving the accuracy of learning with respect to these outputs. This is in contrast to the application of IoAlergia in an active setting by Chen and Nielsen [79], as they aimed at actively improving the overall accuracy of learned models.

Overview of Approach. More concretely, inspired by black-box checking [230], we propose the following learning-based testing approach to analyse reactive systems exhibiting stochastic behaviour in a black-box setting. Instead of targeting general properties, for instance, formulated in probabilistic computation tree logic (PCTL), we check reachability as a first step. Since we follow a simulation-based approach, we check step-bounded reachability. Hence, we focus on formulas of the form $F^{<k}\psi$. The restriction to bounded properties is also common in SMC [177], which is simulation-based and applied in this context. SMC of unbounded properties is actually especially challenging in the black-box setting [187] that we consider in learning.

Rather than learning DFA like Peled et al. [230], we assume that systems can be modelled by MDPs. Hence, we consider systems controllable by inputs, chosen by an environment or a tester. As such, these systems involve non-determinism resulting from the choice of inputs and stochastic behaviour reflected in state transitions and outputs. Given such a system, our goal in bounded reachability checking is to find an input-selection strategy, i.e a resolution of non-determinism, which maximises the probability of reaching a certain property within a bounded number of steps. Properties can, for example, be the observation of desired outputs. A possible application scenario for our technique is stress testing of systems with stochastic failures. We could generate a testing strategy that provokes such failures.

The approach we follow is shown in Figure 6.1. First, we sample system traces randomly. Then, we learn an MDP from these traces via the state-merging-based technique IoAlergia, an adaptation of Alergia [74], which we discussed in Section 5.4 and which was originally described by Mao et al. [198, 199]. Once we learned a hypothesis model $\mathcal{M}_{\mathrm{h1}}$, we use the Prism model checker [174] for a reachability analysis to find the maximal probability of reaching a state satisfying a property ψ in $\mathcal{M}_{\mathrm{h1}}$. Prism computes a probability p and a scheduler s_1 to reach ψ with probability p. Since IoAlergia learns models from system traces, the quality of the model $\mathcal{M}_{\mathrm{h1}}$ depends on these traces. If ψ is not adequately covered, s_1 derived from $\mathcal{M}_{\mathrm{h1}}$ may perform poorly and rarely reach ψ. To account for that,

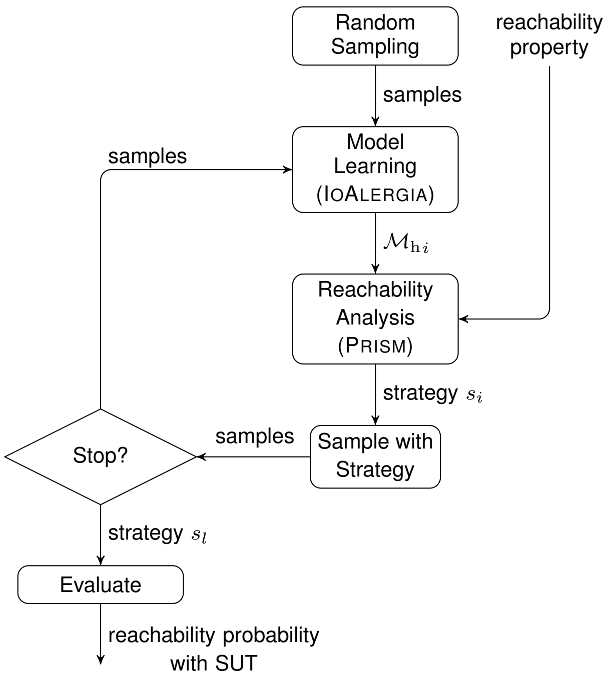


Figure 6.1: Overview of probabilistic black-box reachability checking

we follow an incremental process. After initial random sampling, we iteratively learn models $\mathcal{M}_{\mathrm{h}i}$ from which we derive schedulers s_i. To sample new traces for $\mathcal{M}_{\mathrm{h}i+1}$ we select inputs randomly and based on s_i. In other words, we use the scheduler s_i for directed testing. Selecting inputs with s_i ensures that paths relevant to ψ will be explored more thoroughly. Traces sampled in this way shall improve the accuracy of learned models with respect to ψ such that subsequently derived schedulers reach ψ more frequently. This process is repeated until either a maximal number of rounds n has been executed, or a heuristic detects that the search has converged to a scheduler.

We mainly use Prism to generate strategies, but ignore the probabilities computed in the reachability analysis. Since the computations are based on possibly inaccurate learned models, the probabilities may significantly differ from the true probabilities. Schedulers, however, may serve as testing strategies regardless of the accuracy of the learned models. In fact, we evaluate the final scheduler generated in the process described above via directed testing of the SUT that is assumed to behave like an MDP. Since the behaviour of an MDP under a scheduler is purely probabilistic, this is a form of Monte Carlo simulation, such that we can apply SMC [177]. The evaluation provides an estimation of the probability of reaching ψ with the actual SUT controlled by scheduler s_l, where l is the last round that has been executed. By directly interacting with the SUT during evaluation, the computed estimation is an approximate lower bound for the optimal probability of reachability with the SUT. In contrast to this, the reachability probabilities computed by Prism based on the learned model do not enjoy this property.

In summary, we combine techniques from various disciplines to optimise the probability of observing desired outputs within a bounded number of steps.

Learning: we rely on IoAlergia for learning MDPs. This algorithm has been developed with verification in mind and it has been evaluated in a model-checking context [198, 199].

Probabilistic Model-Checking: we use Prism [174], a state-of-the-art probabilistic model checker, to generate strategies for bounded reachability based on learned models.

Testing: directed sampling guided by a strategy is a form of model-based testing with learned models. The sampling algorithm was developed for the presented technique.

Statistical Model-Checking: we evaluate the final strategy on the SUT. As the SUT is a black-box system, we cannot apply probabilistic model-checking on it and instead perform a Monte Carlo simulation to estimate reachability probabilities, like in SMC [177].

Chapter Structure. Section 6.2 discusses the proposed approach for learning-based testing of stochastic systems in more detail. We present evaluation results in Section 6.3. In Section 6.4, we provide a summary and we conclude this chapter in Section 6.5 by discussing our findings focusing on our research questions.

6.2 Method

This section discusses probabilistic black-box reachability checking in more detail. We begin with a detailed discussion of the test-based reachability checking process, then we comment on convergence and finally conclude with considerations regarding the practical application of probabilistic black-box reachability checking.

6.2.1 Reachability Checking Process

To discuss the test-based reachability checking process, we first give an overview, discussing each of the steps in the process briefly. Subsequently, we provide in-depth descriptions of the process steps. In the remainder of this chapter, we assume that we interact with an MDP $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$ representing the SUT of which we only know the inputs I. For the interaction with $\mathcal{M}$, we assume to be able to reset $\mathcal{M}$ to q_0, to perform inputs on $\mathcal{M}$, and to observe outputs of $\mathcal{M}$.

Basically, we try to find an optimal scheduler for satisfying a given reachability property $\phi = F^{<k}\psi$ with the SUT $\mathcal{M}$. For this purpose, we iteratively sample system traces, learn models from the traces and derive schedulers from the learned models. The derived schedulers also serve to refine the sampling strategy. This process is also shown in Figure 6.1 and includes the following steps.

1. **Create Initial Samples.** This step collects a multiset of system traces through interaction with $\mathcal{M}$ by choosing inputs from I uniformly at random and executing them. Basically, we perform random-walks-based testing via Algorithm 2.4 to create the first batch of samples.

For at most $maxRounds$ **rounds do**:

2.1. **Learn Model.** Given the system traces sampled so far, we use IoAlergia to learn an MDP $\mathcal{M}_{\mathrm{h}i} = \langle Q_{\mathrm{h}}, I, O_{\mathrm{h}}, q_{0\mathrm{h}}, \delta_{\mathrm{h}}, L_{\mathrm{h}}\rangle$, where h stands for hypothesis and $i \in [1 \mathinner{..} maxRounds]$ denotes the current round.

2.2. **Analyse Reachability.** Reachability analysis on $\mathcal{M}_{\mathrm{h}i}$ with Prism [174]: we compute the maximum probability $P_{\mathcal{M}_{\mathrm{h}i}, s_i}(\phi)$ of satisfying ϕ and generate the corresponding scheduler s_i in this analysis. We refer to such schedulers also as *learned schedulers*, since they are derived from learned models.

2.3. **Sample with Scheduler.** We extend the multiset of system traces through property-directed sampling. For that, we choose some inputs with the learned scheduler s_i and we choose some inputs randomly. With increasing number of rounds i, we decrease the portion of random choices.

2.4. **Check Early Stop.** We may stop before executing $maxRounds$ rounds if a stopping criterion is satisfied. This criterion is realised with a heuristic check for convergence. In this check, we basically determine whether several consecutive schedulers behave similarly.

3. **Evaluate.** In a last step, we evaluate the most recent scheduler that we have generated. For this evaluation, we sample system traces again, but avoid choosing inputs randomly. The relative number of traces satisfying ϕ now gives us an estimate for the success probability to satisfy ϕ with $\mathcal{M}$, the black-box SUT, controlled by scheduler s_l, where l is the last round that we executed. A Chernoff bound [177, 223], which is commonly used in SMC, specifies the required number of samples.

Create Initial Samples. In the first step, we sample system traces by choosing input actions randomly according to a uniform distribution. Hence, we sample with a scheduler s_{unif} defined as follows: $\forall q \in Q, s_{\text{unif}} : q \mapsto \mu_{\text{unif}}(I)$ where $\forall i \in I : \mu_{\text{unif}} : i \mapsto \frac{1}{|I|}$. Sampling is further controlled by the length probability p_l and by the batch size n_{batch}, which is the number of traces collected at once. These parameters also affect subsequent sampling. Additionally, we set a *seed*-value for the initialisation of pseudo-random functions that are, for instance, used to select random inputs.

As discussed in Section 5.5.1, we need to choose p_l appropriately to enable checking whether a trace satisfies a step-bounded reachability property. Therefore, we set $p_l(j) = 0$ for $j < k - 1$ where k is the step bound of the property we test for. Due to that, each sampled trace has a prefix that is relevant to the property ϕ. This would not be necessary for learning, but we generally apply this constraint. The length of suffixes, i.e. the trace extensions beyond k, follows a geometric distribution parameterised by $p_{\text{quit}} \in [0, 1]$. Before each step, we stop with probability p_{quit}. Hence, the number of input-output pairs $|t|$ in a trace t is distributed according to $p_l(|t|) = (1 - p_{\text{quit}})^{|t|-k+1} p_{\text{quit}}$ for $|t| \geq k - 1$ and $p_l(|t|) = 0$ otherwise. Both p_{quit} and n_{batch} must be supplied by the user. In the remainder of this section, $\mathcal{S}_i$ denotes the multiset of traces created by the i^{th} sampling step, and $\mathcal{S}_{\text{all}}$ refers to the multiset of all sampled traces. Hence, $\mathcal{S}_{\text{all}}$ is initially set to $\mathcal{S}_1$, containing n_{batch} traces distributed according to $\mathbb{P}^l_{\mathcal{M},s_{\text{unif}}}$, collected by random testing. It is continuously extended in the testing process.

Learn Model. In this step, we learn an MDP $\mathcal{M}_{\text{h}i} = \langle Q_\text{h}, I, O_\text{h}, q_{0\text{h}}, \delta_\text{h}, L_\text{h} \rangle$ from $\mathcal{S}_{\text{all}} = \bigcup_{j \leq i} \mathcal{S}_j$ using IoALERGIA [198, 199], which is an approximate system model. Strictly speaking, we learn an MDP with a partial transition function, which we make input-complete with a function *complete*. The transition function of a learned MDP may be undefined for some state-input pair if there is no corresponding trace in $\mathcal{S}_{\text{all}}$. For this reason, we add transitions to a special sink-state labelled with *dontKnow* for undefined state-input pairs. Once we enter that state, we cannot leave it. The label *dontKnow* is more generally a special output label, which is not part of the original output alphabet.

Formally, we learn $\mathcal{M}'_\text{h} = \langle Q'_\text{h}, I, O'_\text{h}, q_0{}'_\text{h}, \delta'_\text{h}, L'_\text{h} \rangle = \text{IOALERGIA}(\mathcal{S}_{\text{all}}, \epsilon_{\text{ALERGIA}})$ and complete $\mathcal{M}'_\text{h}$ via $\mathcal{M}_{\text{h}i} = \textit{complete}(\mathcal{M}'_\text{h})$ where $Q_\text{h} = Q'_\text{h} \cup \{q_{\text{undef}}\}$, $O_\text{h} = O'_\text{h} \cup \{\textit{dontKnow}\}$, with $\textit{dontKnow} \notin O'_\text{h}$, $q_0{}'_\text{h} = q_{0\text{h}}$, $\delta_\text{h} = \delta'_\text{h} \cup \{(q_{\text{undef}}, i) \mapsto \mathbf{1}_{\{q_{\text{undef}}\}} \mid i \in I\} \cup \{(q, i) \mapsto \mathbf{1}_{\{q_{\text{undef}}\}} \mid q \in Q'_\text{h}, i \in I, \nexists d : (q, i) \mapsto d \in \delta'_\text{h}\}$ and $L_\text{h} = L'_\text{h} \cup \{q_{\text{undef}} \mapsto \textit{dontKnow}\}$.

Following the terminology of active automata learning [37], we refer to $\mathcal{M}_{\text{h}i}$ as the current hypothesis. Input completion via *complete* is required by Definition 5.1, but it does not affect the reachability analysis. We maximise the probability of reaching desired events, therefore generated schedulers will not choose to execute inputs leading to the state q_{undef} labelled *dontKnow*. This is due to the fact that once we reach q_{undef}, we have a probability of zero to observe anything other than *dontKnow* according to our hypothesis model.

Analyse Reachability. Given the current hypothesis learned in the last step, our implementation uses the PRISM model checker [174] to derive a scheduler for satisfying the property ϕ. This is achieved by performing the following steps in a fully automated manner:

1. Translate $\mathcal{M}_{\text{h}i}$ into the PRISM modelling language by encoding

   1.1. states using integers,

1.2. inputs using commands labelled with actions, and

1.3. outputs using labels.

2. Since PRISM only supports scheduler generation for unbounded reachability properties, we pre-process the translated $\mathcal{M}_{\mathrm{h}i}$ further and encode ϕ as unbounded property [69]:

   2.1. We add a step-counter variable steps ranging between 0 and k, where k is the step bound of the examined property.

   2.2. The variable steps is incremented with every execution of an input until the maximal value k is reached. Once $\mathit{steps} = k$, steps is left unchanged.

   2.3. We change ϕ to $\phi' = F(\psi \wedge \mathit{steps} < k)$, thus moving the bound from the temporal operator to the property that should be reached.

3. Finally, we use the *sparse engine* of PRISM to compute the maximum probability $\max_s \mathbb{P}_{\mathcal{M}_{\mathrm{h}i},s}(\phi')$ for satisfying ϕ' and export the corresponding scheduler $s_{\mathrm{h}i}$. This is done by verifying the property `Pmax=?[F(psi & steps < k)]` with PRISM.

These steps actually create an MDP $\mathcal{M}_{\mathit{steps}_i}$ in PRISM. This MDP contains $k+1$ copies (q, st) of each state q of $\mathcal{M}_{\mathrm{h}i}$, one for each value st that the variable steps can take. Note that not all $k+1$ copies of a state are reachable. If q' is reachable from q in $\mathcal{M}_{\mathrm{h}i}$, then $(q', st+1)$ is reachable from (q, st) if $st < k$. If $st = k$, then (q', st) is reachable from (q, st). The target states in $\mathcal{M}_{\mathit{steps}_i}$ for the unbounded reachability property $\phi' = F(\psi \wedge \mathit{steps} < k)$ are all (q, st) with $L(q) \models \psi$ and $st < k$. Furthermore, all (q, st) with $st = k$ are non-target states from which we cannot reach target states, as required by the original bounded reachability property ϕ. Given $\mathcal{M}_{\mathit{steps}_i}$ and the unbounded reachability property ϕ', PRISM exports memoryless deterministic schedulers. These schedulers, however, do not define input choices for all states, but only for states reachable by the composition of scheduler and corresponding model. To account for cases with undefined scheduler behaviour, we use the notation $s_{\mathrm{h}i}(q) = \bot$. It denotes that scheduler $s_{\mathrm{h}i}$ does not define a choice for q. The scheduler $s_{\mathrm{h}i}$ is actually defined for states of $\mathcal{M}_{\mathit{steps}_i}$, therefore we treat $\mathcal{M}_{\mathrm{h}i}$ as being transformed to $\mathcal{M}_{\mathit{steps}_i}$ in the following. This allows us to use $s_{\mathrm{h}i}$ on the hypothesis $\mathcal{M}_{\mathrm{h}i}$.

Sample with Scheduler. Property-directed sampling with learned schedulers aims at exploring parts of the system more thoroughly that have been identified to be relevant to the property. To avoid getting trapped in local probability maxima, we also explore new paths by choosing actions randomly with probability $p_{\mathrm{rand}i}$, where i corresponds to the current round. This probability is decreased in each round to explore more broadly in the beginning and focus on relevant parts in later rounds. Two parameters control $p_{\mathrm{rand}i}$: $p_{\mathrm{start}} \in [0,1]$ defines the initial probability and $c_{\mathrm{change}} \in [0,1]$ specifies an exponential decrease, where $p_{\mathrm{rand}1} = p_{\mathrm{start}}$ and $p_{\mathrm{rand}i+1} = c_{\mathrm{change}} \cdot p_{\mathrm{rand}i}$ for $i \geq 1$.

Basically, we execute both, SUT and hypothesis $\mathcal{M}_{\mathrm{h}i}$, in parallel. The former ensures that we sample traces of the actual system while the latter is necessary because the learned scheduler $s_{\mathrm{h}i}$ is defined for $\mathcal{M}_{\mathrm{h}i}$. Stated differently, we need to simulate the path taken by the SUT on the current hypothesis $\mathcal{M}_{\mathrm{h}i}$. This enables selecting actions with $s_{\mathrm{h}i}$. As $\mathcal{M}_{\mathrm{h}i}$ is an approximation, two scenarios may occur in which we cannot use $s_{\mathrm{h}i}$. In the following scenarios, we default to selecting inputs randomly:

1. The SUT may show outputs not foreseen by $\mathcal{M}_{\mathrm{h}i}$. This may happen if parts of the system structure and not only probabilities have been learned incorrectly. In such cases, we cannot determine the correct state transition in $\mathcal{M}_{\mathrm{h}i}$. Consequently, we cannot use $s_{\mathrm{h}i}$ in subsequent states.

2. By performing random inputs we may follow a path that is not optimal with respect to $\mathcal{M}_{\mathrm{h}i}$ and the reachability property ϕ. Thus, we may enter a state where $s_{\mathrm{h}i}$ is undefined.

Algorithm 6.1 Property-directed sampling

```
Input: $p_{\mathrm{rand}i}, \mathcal{M}_{\mathrm{h}i}, s_{\mathrm{h}i}, n_{\mathrm{batch}}, \phi = F^{<k}\psi, \mathcal{S}_{\mathrm{all}}, c_{\mathrm{change}}$
Output: $p_{\mathrm{rand}i+1}, \mathcal{S}_{i+1}, \mathcal{S}_{\mathrm{all}}$
 1: $\mathcal{S}_{i+1} \leftarrow \{\}$
 2: while $|\mathcal{S}_{i+1}| < n_{\mathrm{batch}}$ do
 3:     $\mathcal{S}_{i+1} \leftarrow \mathcal{S}_{i+1} \cup \{\textsc{sample}(p_{\mathrm{rand}i}, \mathcal{M}_{\mathrm{h}i}, s_{\mathrm{h}i}, k)\}$
 4: end while
 5: $p_{\mathrm{rand}i+1} \leftarrow p_{\mathrm{rand}i} \cdot c_{\mathrm{change}}$
 6: $\mathcal{S}_{\mathrm{all}} \leftarrow \mathcal{S}_{\mathrm{all}} \cup \mathcal{S}_{i+1}$
 7: function SAMPLE($p_{\mathrm{rand}i}, \mathcal{M}_{\mathrm{h}i}, s_{\mathrm{h}i}, k$)
 8:     $trace \leftarrow \mathbf{reset}()$
 9:     $q_{\mathrm{curr}} \leftarrow q_{0\mathrm{h}i}$
10:     while $|trace| - 1 < k \lor \neg coinFlip(p_{\mathrm{quit}})$ do
11:         if $coinFlip(p_{\mathrm{rand}i}) \lor q_{\mathrm{curr}} = \bot \lor s_{\mathrm{h}i}(q_{\mathrm{curr}}) = \bot$ then
12:             $input \leftarrow rSel(I)$
13:         else
14:             $input \leftarrow s_{\mathrm{h}i}(q_{\mathrm{curr}})$
15:         end if
16:         $outSut \leftarrow \mathbf{step}(input)$
17:         $trace \leftarrow trace \cdot (input \cdot outSut)$
18:         $distQ_{\mathrm{curr}} = \delta_{\mathrm{h}i}(q_{\mathrm{curr}}, input)$
19:         $q_{\mathrm{curr}} \leftarrow \begin{cases} q \in Q_{\mathrm{h}i} & \text{such that } L_{\mathrm{h}i}(q) = outSut \land distQ_{\mathrm{curr}}(q) > 0 \\ \bot & \text{if there is no such } q \end{cases}$
20:     end while
21:     return $trace$
22: end function
```

The sampling is detailed in Algorithm 6.1. In addition to artefacts generated by other steps, such as the input-enabled hypothesis[1] $\mathcal{M}_{\mathrm{h}i}$ and the generated scheduler $s_{\mathrm{h}i}$, sampling requires two auxiliary operations:

- $\mathbf{reset}$: resets the SUT to the initial state and returns the unique initial output symbol
- $\mathbf{step}$: executes a single input changing the state of the SUT and returning the corresponding output

Both operations are realised by a test adapter. Lines 1 to 5 of the algorithm collect traces by calling the SAMPLE function and update $p_{\mathrm{rand}i}$. The SAMPLE function returns a single trace, which is created on-the-fly and initialised with the output produced upon resetting the SUT (Line 8). Line 9 initialises the current model state. Afterwards, an input is chosen (Line 11 to Line 14). It is chosen randomly with probability $p_{\mathrm{rand}i}$ or if we cannot determine an input (Line 11 and Line 12). Otherwise, the input is selected with $s_{\mathrm{h}i}$ (Line 14). We record the output of the SUT in response to the input and extend the trace in Line 16 and Line 17. The next two lines update the model state. In case, the SUT produces an output which is not allowed by the model, the new model state is undefined (second case in Line 19). This corresponds to the first scenario, in which we default to choosing inputs randomly. Trace creation stops if the trace is long enough and if a probabilistic Boolean choice returns $\mathbf{true}$ (Line 10). Hence, the actual trace length follows a probability distribution. Note that lines 11 to 19 implement a randomised scheduler derived from $s_{\mathrm{h}i}$. We will refer to this scheduler as $randomised(s_{\mathrm{h}i})$. In the taxonomy of Utting et al. [274], property-directed sampling can be categorised as model-checking-based online testing with a combination of requirements coverage and random input-selection as test-selection criterion.

[1]Note that if we reach the state labelled $dontKnow$ during sampling, the outputs of the hypothesis and the SUT are guaranteed to differ. Due to Condition 1 for random input selection, we continue sampling with random inputs after that.

Evaluate. As a result of the reachability analysis, PRISM calculates a probability of reaching ϕ. This probability, however, is derived from a learned model which is possibly inaccurate. Therefore, the calculated probability may greatly differ from the actual probability of reachability with the SUT. To account for that, we evaluate the scheduler $s_\mathrm{h} = s_{\mathrm{h}l}$, where l is the last round we executed. We do this by sampling a multiset of traces $\mathcal{S}_\mathrm{eval}$, while generally selecting inputs with s_h. For that we execute Algorithm 6.1 with $p_{\mathrm{rand}i} = 0$. This effectively samples traces from the DTMC induced by the composition of the SUT $\mathcal{M}$ and the scheduler $\mathit{randomised}(s_\mathrm{h})$. Since the behaviour of this DTMC is fully probabilistic, we can apply SMC. Hence, we estimate $\mathbb{P}_{\mathcal{M},\mathit{randomised}(s_\mathrm{h})}(\phi)$ by $\hat{p}_{\mathcal{M},s_\mathrm{h}} = \frac{|\{s\in\mathcal{S}_\mathrm{eval}|s\models\phi\}|}{|\mathcal{S}_\mathrm{eval}|}$. To achieve a given error bound ϵ_eval with a given confidence $1 - \delta_\mathrm{eval}$, we compute the required number of samples $|\mathcal{S}_\mathrm{eval}| = n_\mathrm{batch}$ based on a Chernoff bound [223], i.e. we apply Equation (5.2). The estimation provides an approximate lower bound of the maximal reachability probability with the SUT. We consider $\hat{p}_{\mathcal{M},s_\mathrm{h}}$ an approximate lower bound, because we know with confidence $1 - \delta_\mathrm{eval}$ that $\max_s \mathbb{P}_{\mathcal{M},s}(\phi)$ is at least as large as $\hat{p}_{\mathcal{M},s_\mathrm{h}} - \epsilon_\mathrm{eval}$.

Check Early Stop. We have observed that the performance of schedulers usually increases with the total amount of available data. Probability estimations derived with intermediate schedulers showed that schedulers generated in later rounds tend to perform better than those generated in earlier rounds. However, we have also seen that fluctuations in these estimations occur over time. Some schedulers may perform worse than schedulers generated in previous rounds. With increasing number of rounds, these fluctuations generally diminish and the estimations converge. Intuitively, this can be explained by the influence of $p_{\mathrm{rand}i}$ in Algorithm 6.1, which controls the probability of selecting random inputs and which decreases over time. As this probability $p_{\mathrm{rand}i}$ approaches zero, we will almost always select inputs with generated schedulers. This will generally only increase the confidence in parts of the system we have already explored, but will not explore new parts and therefore new schedulers are likely to show similar behaviour to previous ones.

Based on these observations, we developed a heuristic check for convergence. If it detects convergence, we stop the iteration early before executing $\mathit{maxRounds}$ rounds. Two simpler checks actually form the basis of the heuristic. The first, called SIMILARSCHED, basically compares the scheduler generated in the current round to the scheduler from the previous round and returns **true** if both behave similarly. The second check, called CONV builds upon the first and reports convergence if we detect statistically similar behaviour via SIMILARSCHED in multiple consecutive rounds. The rationale behind this is that schedulers should behave alike after convergence. We check for similarity rather than for equivalence, because there may be several optimal inputs in a state and slight variations in transition probabilities in the learned models may lead to the different choices of inputs. Furthermore, we can compare schedulers during sampling by checking whether they would choose the same inputs. This gives us a large number of events as basis for our decision and does not require additional sampling.

The convergence check has three parameters: α_conv controlling the confidence level, an error bound ϵ_conv, and a bound on the number of rounds r_conv. The first two parameters control a statistical test which checks whether two schedulers behave similarly. For this test, we consider Bernoulli random variables E_i for $i \in [2 \mathinner{..} \mathit{maxRounds}]$. E_i is equal to one if two consecutive schedulers $s_{\mathrm{h}i}$ and $s_{\mathrm{h}i-1}$ behave the same, by choosing the same input in some state. E_i is zero otherwise. Let p_{E_i} be the corresponding success probability, that is, the probability of E_i being equal to one. We observe samples of E_i in Line 14 of Algorithm 6.1. Each time we choose an input a with $s_{\mathrm{h}i}$, we also determine which input a' the previous scheduler $s_{\mathrm{h}i-1}$ would have chosen. We record a positive outcome if $a = a'$ and a negative outcome otherwise.

Let $\hat{p}_{E_i}$ be the relative number of recorded positive outcomes, which is an estimate of p_{E_i}. If p_{E_i} is equal to one, then both schedulers behave equivalently, as they always choose the same input. Consequently, we test whether $\hat{p}_{E_i}$ is close to one. We test the null hypothesis $H_0 : p_{E_i} \leq 1 - \epsilon_\mathrm{conv}$ against the alternative hypothesis $H_1 : p_{E_i} > 1 - \epsilon_\mathrm{conv}$ with a confidence level of $1 - \alpha_\mathrm{conv}$ using an exact binomial test. The hypothesis H_1 denotes that the compared schedulers choose the same inputs in most of the cases. Let $\textsc{similarSched}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i)$ be the result of this test in round i, which is $\mathbf{true}$ if H_0 is rejected and $\mathbf{false}$ otherwise.

Finally, we can formulate the complete convergence check $\textsc{conv}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i)$. It returns $\mathbf{true}$ in round i if r_{conv} consecutive calls of $\textsc{similarSched}$ returned $\mathbf{true}$, thus

$$\textsc{conv}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, i) = \bigwedge_{j=i-r_{\text{conf}}+1}^{i} \textsc{similarSched}(\alpha_{\text{conv}}, \epsilon_{\text{conv}}, j).$$

Note that $p_{\text{rand}i}$ implicitly affects the convergence check. We collect samples of E_i in Line 14 of Algorithm 6.1, and large $p_{\text{rand}i}$ cause Line 13 to be executed infrequently. As a result, sample sizes of E_i are small in early rounds with large $p_{\text{rand}i}$. This influence on the convergence check is beneficial, because schedulers are more likely to improve if $p_{\text{rand}i}$ is large, as new parts of the system may be explored via frequent random steps.

While the check introduces further parameters, it may simplify the application of the approach in scenarios where we have little knowledge about the system at hand. In such cases, it may be difficult to find a reasonable choice for the number of rounds $maxRounds$. With this heuristic, it is possible to choose $maxRounds$ conservatively, but stop early once convergence is detected. However, it may also impair results, if convergence is detected too early.

6.2.2 Convergence to the True Model

Generally, Mao et al. [199] showed convergence in the large sample limit for IoAlergia. However, the sampling mechanism applied for generating traces needs to ensure that sufficiently many executions of all inputs in all states are observed. This is also discussed in earlier work by Mao et al. [198]. The authors state that IoAlergia requires a *fair* scheduler, one that chooses each input infinitely often. The uniformly randomised scheduler s_{unif} satisfies this requirement. As a result, we have convergence in the limit, if we perform only a single round of learning, in which we sample with s_{unif}.

Property-directed sampling favours certain inputs with increasing number of rounds, but it also selects random inputs with probability $p_{\text{rand}i}$ in round i. If we ensure that $p_{\text{rand}i}$ is always non-zero, we will select all inputs infinitely often in an infinite number of rounds. Therefore, the learned models will converge to the true model (up to bisimulation equivalence) and the derived schedulers will converge to an optimal scheduler.

An alternative way to approach convergence is to follow a hybrid approach by collecting traces via property-directed sampling and via uniform sampling in parallel. Uniform sampling ensures that all inputs are executed sufficiently often, which entails convergence. Property-directed sampling explores parts of the system that have been identified to be relevant, which helps to correctly learn those parts. As a result, intermediate schedulers are more likely to perform well.

Hence, we have convergence in the limit under certain assumptions. In practice, when we learn from limited data, uniform schedulers are likely to be insufficient if events occur only after long interaction scenarios. If events occur rarely in the sampled system traces, then it is unlikely that the part modelling those events is accurately learned. Active learning, as described by Chen and Nielsen [79], addressed this issue by guiding sampling so as to reduce the uncertainty in the learned model. In Chapter 7, we also actively guide sampling towards rarely observed behaviour of stochastic SULs. The approach discussed in this chapter similarly guides sampling, but with the aim at reducing uncertainty along traces that are likely to satisfy a reachability property.

As noted above, we have seen that the learned schedulers usually converge to a scheduler. This scheduler may not be globally optimal, though. We also performed experiments with the outlined hybrid approach to avoid getting trapped in local maxima by collecting half of the system traces through uniform sampling. While it showed favourable performance in a few cases, the non-hybrid approach generally

Table 6.1: All parameters with short descriptions

Parameter	Description
n_{batch}	number of traces sampled in one round
$maxRounds$	maximum number of rounds
p_{start}	initial probability of random input selection
c_{change}	factor changing the probability of random input selection
p_{quit}	parameter of geometric distribution of sampled trace length
$\epsilon_{\text{ALERGIA}}$	controls significance level of statistical compatibility check of IoALERGIA
$1 - \alpha_{\text{conv}}$	confidence level of convergence check
ϵ_{conv}	error bound of convergence check
$1 - \delta_{\text{eval}}$	confidence level of scheduler evaluation (Chernoff bound)
ϵ_{eval}	error bound of scheduler evaluation (Chernoff bound)
r_{conv}	number of rounds considered in convergence check

produced better results with the same number of samples. Therefore, we will not discuss experiments with the hybrid approach.

Apart from convergence, it may not always be necessary to find a (near-)optimal scheduler. A requirement may state that the probability of reaching an erroneous state e must be smaller than some p. By learning and evaluating a scheduler s_{h} such that the probability estimate $\hat{p}_{\mathcal{M},s_{\text{h}}}$ of reaching e satisfies $\hat{p}_{\mathcal{M},s_{\text{h}}} \geq p$, we basically show with some confidence that the requirement is violated. Such a requirement could be the basis of another stopping criterion. If in round i, a sufficiently large number of the sampled traces $\mathcal{S}_i$ reaches an erroneous state, we may decide to evaluate the corresponding scheduler $s_{\text{h}i-1}$. We could then stop if $\hat{p}_{\mathcal{M},s_{\text{h}i-1}} \geq p$ and continue otherwise.

6.2.3 Application and Choice of Parameters

We will now briefly discuss the choice of parameters taking our findings into account. A summary of all parameters along with a concise description is given in Table 6.1.

The product $n_{\text{s}} = n_{\text{batch}} \cdot maxRounds$ defines the overall maximum number of samples for learning, thus it can be chosen as large as the testing/simulation budget permits. Increasing $maxRounds$ while fixing n_{s} increases the time required for learning and model checking. Intuitively, it improves accuracy as well, because sampling is more frequently adjusted towards the considered property. For the systems examined in Section 6.3, values in the range between 50 and 200 led to reasonable accuracy while incurring an acceptable runtime overhead. Runtime overhead is the time spent learning and model checking, as opposed to the time spent doing actual testing, i.e. (property-directed) sampling. The convergence check takes three parameters as input for which we identified well-suited default parameters. To ensure a high confidence of the statistical test, we set $\alpha_{\text{conv}} = 0.01$. Since schedulers should choose the same input in most cases, ϵ_{conv} should be small, but greater than zero to allow for some variation. In our experiments, we set it to $\epsilon_{\text{conv}} = 0.01$ and we set $r_{\text{conv}} = 6$. More conservative choices would be possible at the expense of performing additional rounds.

The value of p_{start} should generally be larger than 0.5, while c_{change} should be close to 1. This ensures broad exploration in the beginning and more directed exploration afterwards. Finally, the choice of p_{quit} depends on the simulation budget and the number of inputs. If there is a large number of inputs, it may be highly improbable to reach certain states within a small number test steps via random testing. Consequently, we should allow for the execution of long tests, in order to reach states requiring complex combinations of inputs. Domain knowledge may also aid in choosing a suitable p_{quit}. If we, for instance, expect a long initialisation phase, p_{quit} should be low to ensure that we reach states following the initialisation.

6.3 Experiments

We evaluated our learning-based testing approach in five case studies from the areas of automata learning, control policy synthesis, and probabilistic model-checking. For the first case study, we created our own model of the slot machine described by Mao et al. [199] in the context of learning MDPs. Two case studies consider models of network protocols extended with stochastic failures. For that, we transformed deterministic Mealy-machine models as detailed below. The model used in the fourth case study is inspired by the gridworld example, for which Fu and Topcu synthesised control strategies [116]. Finally, we generate schedulers for a consensus protocol [42] which serves as a benchmark in probabilistic model-checking.

Adding Stochastic Failures. Deterministic Mealy machines serve as the basis for two case studies. These Mealy machines model communication protocols and are the results from previous learning experiments. One of the Mealy machines is part of the TCP models learned by Fiterău-Broştean et al. [112] and we also use one of the MQTT models that we learned with the setup described in Chapter 3 [263]. Basically, we simulate stochastic failures by adding outputs represented by the label $crash$. These occur with a predefined probability instead of the correct output. Upon such a failure, the system is reset. We implemented this by transforming the Mealy machines as follows:

1. Translate a Mealy machine into a Moore machine, creating an MDP $\mathcal{M} = \langle Q, I, O, q_0, \delta, L \rangle$ with a non-probabilistic δ.
2. Extend O with a new symbol $crash$ and add a new state q_{cr} to Q with $L(q_{\text{cr}}) = crash$.
3. For a predefined probability p_{cr} and for all o in a predefined set $Crashes$:
   - 3.1. Find all $q, q' \in Q$, $i \in I$ such that $\delta(q,i)(q') = 1$ and $L(q') = o$
   - 3.2. Set $\delta(q,i)(q') = 1 - p_{\text{cr}}$ and $\delta(q,i)(q_{\text{cr}}) = p_{\text{cr}}$
   - 3.3. For all $i \in I$ set $\delta(q_{\text{cr}},i)(q_{\text{cr}}) = p_{\text{cr}}$ and $\delta(q_{\text{cr}},i)(q_0) = 1 - p_{\text{cr}}$

This simulates stochastic failures of outputs belonging to a set $Crashes$. Instead of producing the correct outputs, we output $crash$ and reach state q_{cr} with a certain probability. With the same probability we stay in this state and otherwise we reset the system to q_0 after the crash.

6.3.1 Measurement Setup and Criteria

We have complete information about all models. This allows us to compare our results to optimal values. Nevertheless, for the evaluation we simulate the models and treat them as black-box systems. The state spaces of the models without step-counter variables for bounded reachability are of sizes 471 (slot machine), 63 (MQTT), 157 (TCP), 35 (gridworld), and 272 (consensus protocol), respectively[2]. For each of these systems, we identified an output relevant to the application domain and applied the presented technique to reach states emitting this output within varying numbers of steps. The slot machine grants prizes, therefore we generated strategies to observe the rarest prize. We seeded stochastic failures into the models of MQTT and TCP using the steps discussed above. For this reason, we generated schedulers to reach these failures. The gridworld we used in the evaluation contains a dedicated *goal* location that served as a reachability objective. In case of the consensus protocol, we generated strategies to finish the protocol, i.e. reach consensus, with high probability.

For a black-box MDP $\mathcal{M}$ and a reachability property ϕ, we compare four approaches to find schedulers s for $\mathbb{P}_{\mathcal{M},s}(\phi)$:

[2]The slot machine could actually be modelled with about 160 states [199], but the other models are approximately minimal.

Incremental Scheduler Learning. We apply the incremental approach discussed in Section 6.2 with a fixed number of rounds. Learned schedulers are denoted by s_{inc}.

Incremental with Convergence Check. We apply the incremental approach, but stop if we either detect convergence with CONV or if $maxRounds$ rounds have been executed. Learned schedulers are denoted by s_{conv}.

Monolithic Scheduler Learning. To check if the incremental refinement of learned models pays off, we use the same approach as for incremental learning, but set $maxRounds = 1$. In other words, we sample traces by solely choosing inputs randomly. Based on this, we perform a single round, which includes learning a model and deriving a scheduler that we evaluate. To balance the simulation budget, we collect $maxRounds \cdot n_{\text{batch}}$ traces, where $maxRounds$ and n_{batch} are the parameter settings for learning s_{inc}. We denote monolithically learned schedulers by s_{mono}.

Uniform Schedulers. As a baseline for comparison, we compare learned schedulers to the randomised scheduler s_{unif} which chooses inputs according to a uniform distribution. This resembles random testing without additional knowledge.

Furthermore, let $s_{\text{opt}} = \text{argmax}_s \mathbb{P}_{\mathcal{M},s}(\phi)$ be the optimal scheduler for a given SUT $\mathcal{M}$ and a property ϕ. As the most important measure of quality, we compare estimates of $\mathbb{P}_{\mathcal{M},s}(\phi)$ to the maximal probability $\mathbb{P}_{\mathcal{M},s_{\text{opt}}}(\phi)$. We consider a scheduler s to be *near optimal*, if the estimate $\hat{p}_{\mathcal{M},s}$ of $\mathbb{P}_{\mathcal{M},s}(\phi)$ derived via SMC is approximately equal to $\mathbb{P}_{\mathcal{M},s_{\text{opt}}}(\phi)$, i.e. $|\hat{p}_{\mathcal{M},s} - \mathbb{P}_{\mathcal{M},s_{\text{opt}}}(\phi)| \leq \epsilon$, for an $\epsilon > 0$. In the following, we use $\epsilon = \epsilon_{\text{eval}}$ for deciding near optimality, where ϵ_{eval} is the error bound of the applied Chernoff bound (Equation (5.2)) [223].

We balance the number of test steps for the incremental and the monolithic approach by executing the same number of tests. As a result, the simulation costs for executing tests is approximately the same. Since the incremental approach requires model learning and model checking in each round, it will also require more computation time than the monolithic approach. While we focus on evaluating with respect to the achieved probability estimation, we will briefly discuss computation costs at the end of the section.

We also briefly discuss estimations based on model checking of learned models $\mathcal{M}_{\text{h}}$. For that, we calculate $\max_s \mathbb{P}_{\mathcal{M}_{\text{h}},s}(\phi)$ with PRISM [174]. These estimations have also been discussed by Mao et al. [199]. They noted that estimations may differ significantly from optimal values in some cases, but generally represent good approximations.

Implementation and Settings. The evaluation is based on our Java implementation of the presented technique which can be found on GITHUB [259]. All experiments were performed on a Lenovo Thinkpad T450 with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz and running Xubuntu Linux 18.04. The systems were modelled with PRISM [174]. PRISM served three purposes:

- We exported the state, transition, and label information from PRISM models. We simulated the models in a black-box fashion with this information.
- The maximal probabilities were computed via PRISM.
- PRISM's scheduler generation was used to derive schedulers.

Simulation as well as sampling is controlled by probabilistic choices. To ensure reproducibility, we used fixed seeds for pseudo-random number generators controlling the choices. All experiments were run with 20 different seeds and we discuss statistics derived from 20 such runs. For the evaluation of schedulers, we applied a Chernoff bound with $\epsilon_{\text{eval}} = 0.01$ and $\delta_{\text{eval}} = 0.01$. We used a fixed significance level for the compatibility check of IOALERGIA, by setting $\epsilon_{\text{ALERGIA}} = 0.5$, a value also used by Mao et al. [199]. They noted that IOALERGIA is generally robust with respect to the choice of this value, but also suggested a lower, data-dependent value for $\epsilon_{\text{ALERGIA}}$ as an alternative. We found that our approach

Table 6.2: General parameter settings for experiments

Parameter	Value
p_{start}	0.75
c_{change}	0.95
$\epsilon_{\text{ALERGIA}}$	0.5
α_{conv}	0.01
ϵ_{conv}	0.01
δ_{eval}	0.01
ϵ_{eval}	0.01
r_{conv}	6

Table 6.3: Parameter settings for the slot-machine case study

Parameter	Value
p_{quit}	0.05
$\mathit{maxRounds}$	100
n_{batch}	1000

benefits from a larger $\epsilon_{\text{ALERGIA}}$, which causes fewer state merges and consequently larger models. Put differently, our approach benefits from more conservative state merging.

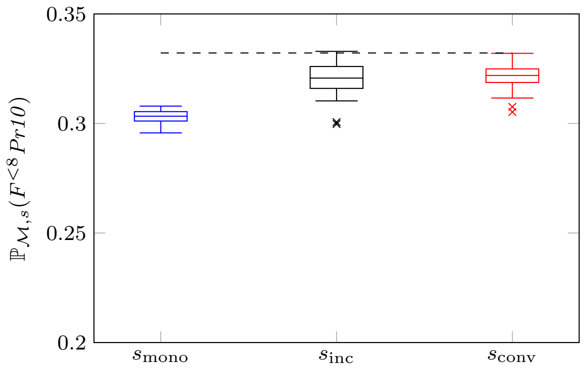
As noted in Section 6.2, we should ensure broad exploration in the beginning and property-directed exploration in later rounds. Therefore, we set $p_{\text{start}} = 0.75$ and $c_{\text{change}} = 0.95$ unless otherwise noted. We set the convergence-check parameters in all experiments as suggested in Section 6.2 to $\alpha_{\text{conv}} = 0.01$, $\epsilon_{\text{conv}} = 0.01$, and $r_{\text{conv}} = 6$. Table 6.2 summarises parameter settings that apply in general.

6.3.2 Slot-Machine Experiments

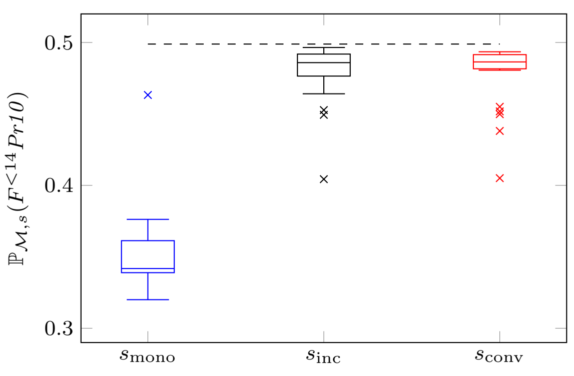
The slot machine was analysed in the context of MDP learning before [199]. Basically, it has three reels which are initially blank and which either show *apple* or *bar* after spinning (one input per reel). With increasing number of spins the probability of bar decreases. A player is given a number of spins m, after which one of three prizes is awarded depending on the reel configuration. A fourth input leads with equal probability either to two extra spins (with a maximum of m), or to stopping the game prematurely including issuance of prizes. For the evaluation, we reimplemented the model, therefore the probabilities and the state space differ from the model by Mao et al. [199]. As property, we investigated reaching the output $\mathit{Pr10}$ if $m = 5$, representing a prize that is awarded after stopping the game if all reels show bar. The parameter settings for the learning experiments are given by Table 6.3, which specifies $p_{\text{quit}} = 0.05$, $\mathit{maxRounds} = 100$, and $n_{\text{batch}} = 1000$.

Figure 6.2 shows evaluation results comparing the different approaches. Box plots summarising the probability estimations for reaching $\mathit{Pr10}$ in less than 8 steps are shown in Figure 6.2a and Figure 6.2b shows results for a limit of 14 steps. From left to right, the blue boxes correspond to s_{mono}, the black boxes correspond to s_{inc}, and the red boxes correspond to s_{conv}, which is the incremental approach with convergence check. Dashed lines mark optimal probabilities. Note that estimations may be slightly larger than the optimal value in rare cases because they are based on simulations. This can be observed for s_{inc} in Figure 6.2a and also in some of the following experiments. The applied Chernoff bound gives a confidence value for staying within error bound ϵ_{eval}, in case we actually find an optimal scheduler.

Estimations with the baseline s_{unif} are fairly constant, at approximately 0.012 for 8 steps and at 0.019 for 14 steps. As estimations with s_{mono}, s_{inc}, and s_{conv} are significantly higher, this shows that our approach positively influences the probability of reaching a desired event. We further see that the incremental approach performs better than the monolithic, whereby the gap increases with step size.

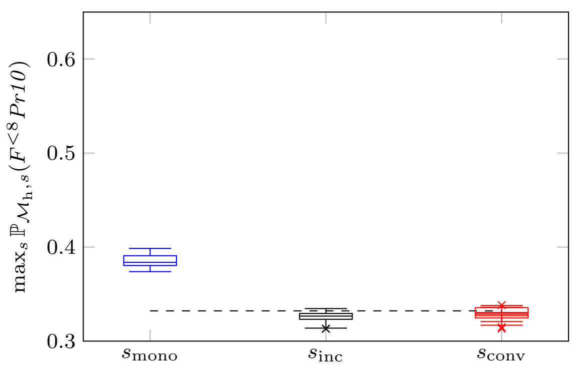


(a) Reaching *Pr10* in less than 8 steps

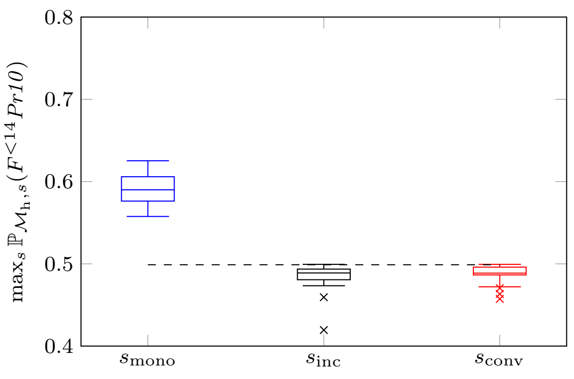


(b) Reaching *Pr10* in less than 14 steps

Figure 6.2: Simulation-based probability estimations of reaching *Pr10* with the slot machine



(a) Reaching *Pr10* in less than 8 steps



(b) Reaching *Pr10* in less than 14 steps

Figure 6.3: Model-checking-based probability estimations of reaching *Pr10* with the slot machine

Unlike the monolithic approach, the incremental approach finds near-optimal schedulers in both cases. However, the relative number of near-optimal schedulers decreases with increasing step bound.

Early stopping via detecting convergence affects performance only slightly. The differences between the quartiles derived for s_{conv} and for s_{inc} are actually smaller than the error bound $\epsilon_{\text{eval}} = 0.01$. Random variations could therefore be the cause of a visually perceived performance change. For the first experiment with a limit of 8 steps, early stopping reduced the number of executed rounds to 72.2 on average. The mean number of rounds performed in the second experiment was reduced to 71.05. However, one run of the first experiment failed to stop early, because convergence was not detected in less than 100 rounds. Runs of the second experiment executed at most 92 rounds.

Alternatively to simulation-based estimation, estimations may be based on model checking a learned model [199]. For that, a model $\mathcal{M}_{\text{h}}$ is learned , either incrementally or in a single step, and then a probabilistic model-checker computes $\max_s P_{\mathcal{M}_{\text{h}},s}(\phi)$. In other words, SMC of the actual SUT controlled by a learned scheduler is replaced by probabilistic model-checking of a learned model. In the first scenario, estimations are generally bounded above by the optimal probability while estimations in the second scenario may be larger than the true optimal probability. An advantage of the model-checking-based scenario is that it reduces the simulation cost since SMC requires additional sampling of the SUT.

Figures 6.3a and 6.3b show model-checking-based estimations of reaching *Pr10* in less than 8 and 14 steps respectively. Here, s_{mono} denotes that the models $\mathcal{M}_{\text{h}}$ were learned in one step, while s_{inc} denotes incremental model learning. Incremental model learning with early stopping is labelled s_{conv}. The figures demonstrate that these estimations differ from estimations obtained via SMC (see Figure 6.2).

Table 6.4: Parameter settings for the MQTT case study

Parameter	Value
p_{quit}	0.025
$maxRounds$	60 (240 for s_{conv})
n_{batch}	100

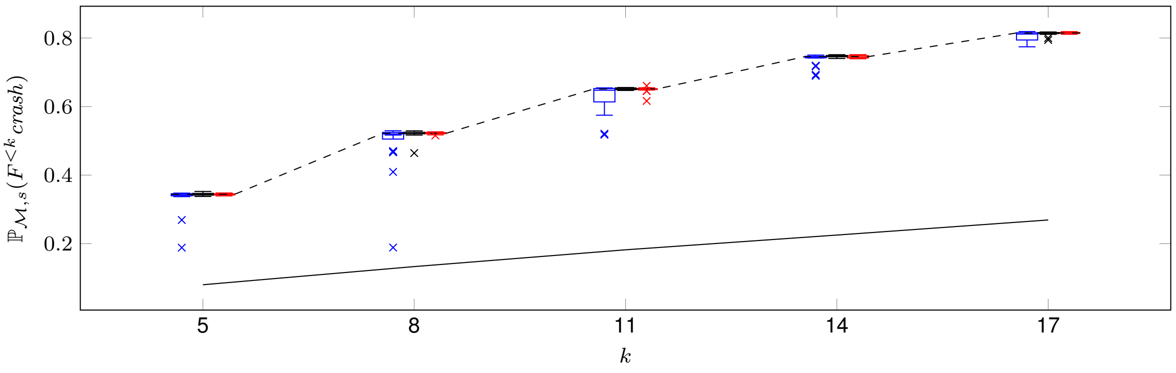


Figure 6.4: Box plots of probability estimations of different learning configurations for MQTT

The monolithic approach computes estimations that are significantly larger than the true optimal value in both cases. The incremental approach leads to more accurate results. None of the measurement results exceeds the optimal value by more than ϵ_{eval}. Note that early stopping did not significantly affect these estimations. Still, the SMC-based estimations are more reliable in the sense that they establish an approximate lower bound on the true optimal probability.

6.3.3 MQTT Experiments

The following case study is based on a Mealy-machine model of an MQTT [73] broker, learned in our case study on learning-based testing MQTT presented in Chapter 3 [263]. We transformed a model of the EMQ[3] broker learned with the mapper *Two Clients with Retained Will*, adding stochastic failures to connection acknowledgements and to subscription acknowledgements for the second client, where we set $p_{\text{cr}} = 0.1$. For the evaluation we learn schedulers s maximising $\mathbb{P}_{\mathcal{M},s}(F^{<k} crash)$ for $k \in \{5, 8, 11, 14, 17\}$. For the sampling, we set $p_{\text{quit}} = 0.025$. This leads to samples longer than necessary for evaluation, because, for instance, for $k = 5$ the expected length of traces is 43. However, it increases the chance of seeing *crash* in a sample which is reflected in learned models. The simulation budget is limited by $maxRounds = 60$ and $n_{\text{batch}} = 100$ for the incremental approach without early stopping. Since the experiments required more than 60 rounds for convergence to be detected, we set $maxRounds$ to 240 for the incremental approach with convergence check. The parameters are also summarised in Table 6.4.

Figure 6.4 shows box plots for the scheduler-generation approaches. At each k, the box plots from left to right summarise measurements for s_{mono} (blue), s_{inc} (black), and s_{conv} (red). The dashed line is the optimal probability achieved with s_{opt}, and the solid line represents the average probability of reaching *crash* with a uniformly randomised scheduler. The box plots demonstrate that larger probabilities are achievable with learning-based approaches than with random testing. All runs including outliers reach *crash* with a higher probability than random testing. The monolithic approach, however, only performs marginally better in some cases. All learning-based approaches find at least one near-optimal scheduler in 20 learning runs, but incremental learning finds near-optimal schedulers more reliably than monolithic scheduler generation.

[3]Website: `https://www.emqx.io/`, accessed on November 4, 2019 – this broker was called emqtt(d) when we performed our MQTT case study presented in Chapter 3.

Table 6.5: Parameter settings for the TCP case study

Parameter	Value
p_{quit}	0.025
maxRounds	120 (240 for s_{conv})
n_{batch}	250

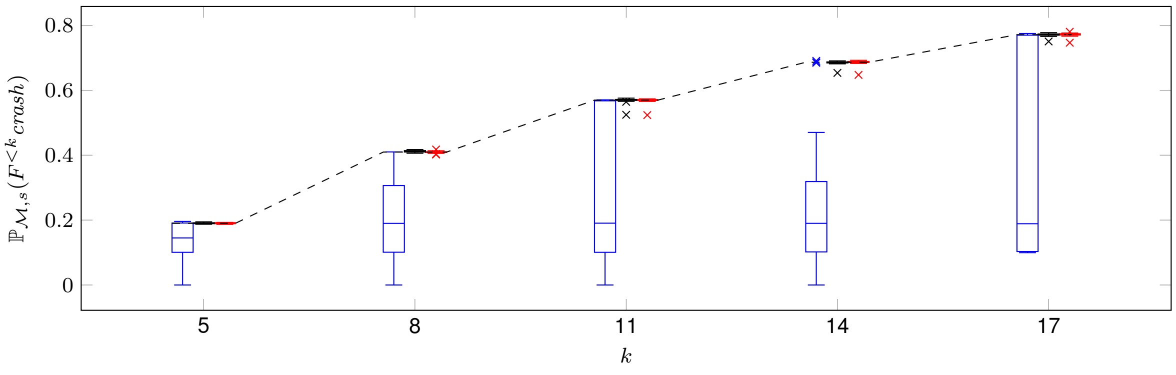


Figure 6.5: Box plots of probability estimations of different learning configurations for TCP

The convergence check causes a reliability gain for $k = 8$ and $k = 17$ in this case study, as it basically detected that executing 60 rounds is not enough. It was generally required to perform more than 60 rounds to detect convergence, except in a few cases. Experiments for larger values of k required slightly more rounds to be executed, such that on average 79.6 rounds were executed for $k = 17$. In contrast to this, we executed on average only 72.15 for $k = 5$. We also see that most estimations of s_{inc} and s_{conv} are in a small range near to the optimal values. However, a few outliers are significantly lower. For instance, one measurement for $k = 8$ is at 0.46. Therefore, it makes sense to learn multiple schedulers and discard those performing poorly.

Model-checking-based estimations of reaching *crash* with the incremental approach led to overestimations in some cases. For instance, the maximal estimation for $k = 11$ is 0.724 while 0.651 is the true optimal value. Also for $k = 5$, one run leads to a model-checking-based estimation of 0.373 although 0.344 is the true optimal value. This is in contrast to the slot machine example (see Figure 6.3), where the incremental approach produced model-checking-based results close to or lower than the optimal value.

6.3.4 TCP Experiments

The TCP case study is based on a Mealy-machine model of Ubuntu's TCP server learned by Fiterău-Broştean et al. [110, 112]. In Section 4.4 [15, 17], we have shown that conformance testing of this system is challenging. Here, we consider a version with random crashes with $p_{\mathrm{cr}} = 0.05$, as discussed in the beginning of this section. We mutated transitions to states outputting an acknowledgement which increments both sequence and acknowledgement numbers. For the evaluation, we learn schedulers for $\mathbb{P}_{\mathcal{M},s}(F^{<k}crash)$ with $k \in \{5, 8, 11, 14, 17\}$ and we set $maxRounds = 120$, and $n_{\mathrm{batch}} = 250$ for the incremental approach without early stopping. Consequently, we set $n_{\mathrm{batch}} = 250 \cdot 120$ for the monolithic approach. Since the convergence check detected convergence only after 120 rounds in several experiments, we set *maxRounds* to 240 for the incremental approach with early stopping. We set p_{quit} to the same value as for MQTT. The parameter settings are also shown in Table 6.5.

Figure 6.5 shows box plots summarising the computed probability estimations. As before, there are groups of three box plots at each k, which from left to right represent s_{mono}, s_{inc}, and s_{conv}. The figure does not include plots for random testing with s_{unif}, because it reaches the crash with very low probability. Estimations produced by s_{unif} are lower than 0.01 for all k. This demonstrates that random testing is insufficient in this case to reliably reach crashes of the system.

We further see that all learning-based approaches achieve to generate near-optimal schedulers for all k. As before, both configurations of the incremental approach are more reliable than the monolithic approach. For this more complex system, the reliability gain from incremental scheduler generation is actually much larger than for the MQTT experiments. Early stopping affects probability estimations only marginally. This is also in line with previous observations.

Like for MQTT, we needed to set $maxRounds$ to a value larger than initially planned, for convergence to be detected. There is a large spread in the number of executed rounds. We, for instance, executed between 42 and 240 rounds for $k = 14$. In this case, convergence was detected after 133.5 rounds on average. The average number of executed rounds is lower than 135 rounds for all k.

6.3.5 Gridworld Experiments

The following case study is inspired by a motion-planning scenario discussed by Fu and Topcu [116], also in the context of learning control strategies. In the experiments, we generate schedulers for a robot navigating in a gridworld environment. These schedulers shall with high probability reach a fixed goal location after starting from a fixed initial location.

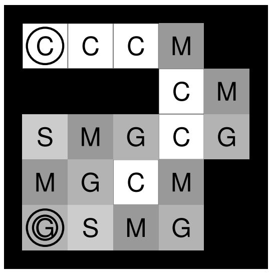
A gridworld consists of tiles of different terrains and is surrounded by walls. To model obstacles, interior tiles may be walls as well. The robot starts at a predefined location and may move into one of four directions via one of four inputs. It can observe changes in the type of terrain, whether it bumped into a wall, and whether it is located at the goal location. If the robot bumps into a wall, it will not change location. Whenever the robot moves, it may not reach its target, but rather reach a tile neighbouring the target with some probability, unless the neighbouring tile is a wall. For example, if the robot moves north, it may reach the tile situated north west or north east to its original position. The probability of such an error depends on the terrain of the target tile. We distinguish the terrains (with error probabilities in parentheses): *Mud* (0.4), *Sand* (0.25), *Concrete* (0), and *Grass* (0.2). As indicated above, *Wall* is actually also a terrain that cannot be entered.

We created the gridworld such that adjacent tiles, with non-zero error probabilities, do not have the same terrain. Otherwise, the MDP modelling the gridworld would be non-deterministic, because it would contain indistinguishable, but different states, reached by the same input. If the generating MDP is non-deterministic, it cannot be guaranteed that IoAlergia is able to learn an adequate deterministic approximation [198].

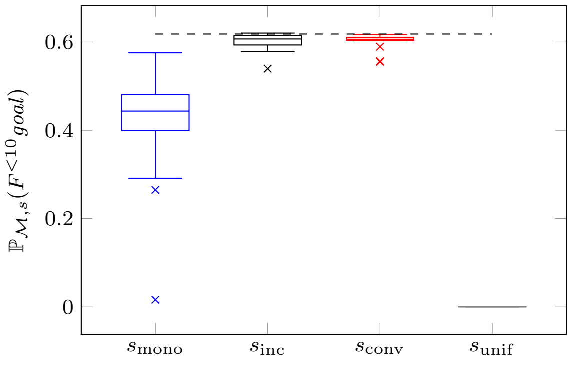
Figure 6.6a shows the gridworld that we used for evaluation. Black tiles represent walls, while the other terrains are represented by different shades of grey and their initial letters. A circle marks the initial location and a double circle marks the goal location. Although its state space, containing 35 different states[4], is relatively small, navigating in this gridworld is challenging without prior knowledge. Initially, three moves to the right are necessary, as walls block direct moves towards the goal. This mimics the requirement of performing an initialisation routine.

Before discussing measurements, we want to briefly describe the structure of the MDP modelling this gridworld and how probabilities affect it. The initial state is labelled C and corresponds to the location with the coordinate $(1, 1)$. Moving towards north is not possible, therefore the input $north$ leads to a state labelled $Wall$, which also corresponds to the coordinate $(1, 1)$. If the robot instead moves towards east, it will reach a state corresponding to the coordinate $(2, 1)$, which is labelled C. Moving from coordinate $(3, 1)$ towards east, the target tile with the coordinate $(4, 1)$ is labelled by M (Mud). This tile has a non-zero error probability of 0.4, therefore the input $east$ causes a stochastic transition; with a probability of 0.6 the robot reaches the target coordinate $(4, 1)$ and observes M, but with a probability of 0.4, it reaches the location $(4, 2)$ to the south and observes C.

[4]The number of states does not equal the number of reachable locations because locations adjacent to walls require two states in the MDP – one outputting the terrain and one with the output $Wall$. States outputting $Wall$ are reached after the robot bumps into a wall.



(a) The gridworld used in our evaluation



(b) Reaching *goal* in the gridworld in less than 10 steps

Figure 6.6: The evaluation gridworld and corresponding experimental results

Table 6.6: Parameter settings for the gridworld case study

Parameter	Value
p_{quit}	0.5
$maxRounds$	150
n_{batch}	500
c_{change}	0.975

To learn schedulers, we applied the configuration given by Table 6.6, where $maxRounds = 150$, and $n_{\text{batch}} = 500$, and $p_{\text{quit}} = 0.5$. Due to the larger value of $maxRounds$, we increased c_{change} as well to 0.975. This causes more random choices and hence broad exploration in a larger number of rounds. As this case study differs significantly from the others, we chose $maxRounds$ conservatively, performing a larger number of rounds.

Figure 6.6b shows measured estimations of $\mathbb{P}_{\mathcal{M},s}(F^{<10}goal)$ for s_{inc}, s_{conv}, s_{mono}, and random testing with s_{unif}. The dashed line denotes the optimal probability.

Random testing obviously fails to reach the goal in less than ten steps. This is caused by the fact that it is unlikely to navigate past the walls via random exploration. The performance of the monolithic approach is also affected by this issue, because it learns solely from uniformly randomised traces sampled by s_{unif}. Random exploration covers only the initial part of the state space thoroughly. Therefore, the monolithically generated schedulers tend to perform worse than the incrementally generated schedulers. By directing exploration towards the goal, the incremental approach achieves to generate near-optimal schedulers.

We also see that the impact of the convergence check is not severe. Both settings, with and without convergence check, produced similar results. The convergence check was able to reduce simulation costs for all but three runs of the experiment, in which convergence was not detected in less than 150 rounds. The incremental scheduler generation with early stopping required a minimum of 94 rounds and a mean of 131.9 rounds to detect convergence.

6.3.6 Shared Coin Consensus-Protocol Experiments

The last case study examines scheduler generation for a randomised consensus protocol by Aspnes and Herlihy [42]. In particular, we used a model of the protocol distributed with the PRISM model checker [174] as a basis for this case study.[5] Note that we changed the functionality of the protocol only

[5]A thorough discussion of the consensus protocol model and related experiments can be found at `http://www.prismmodelchecker.org/casestudies/consensus_prism.php`, accessed on November 4, 2019

Table 6.7: Parameter settings for the consensus-protocol case study

Parameter	Value
p_{quit}	0.025
$maxRounds$	100
n_{batch}	250

slightly by performing minor adaptions such as adding action labels for inputs.

This protocol's goal is to achieve consensus between multiple asynchronous processes. This is modelled by finding a common preferred value of either 1 or 2 on which all processes agree. In the model distributed with PRISM, the processes share a global counter c with a range of $[0\,..\,2 \cdot (K+1) \cdot N]$ where N is the number of processes and K is an integer constant. Initially, c is set to $(K+1) \cdot N$. All involved processes perform the following steps to locally determine a preferred value v:

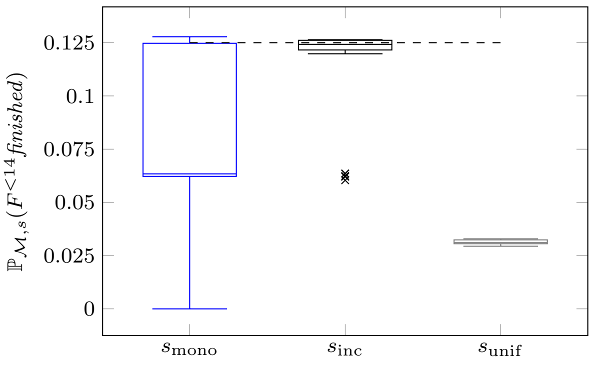
1. Flip a fair coin (local to the process)
2. Check coin
   - 2.1. If the coin shows tails, decrement shared counter c
   - 2.2. Otherwise increment c
3. Check value of c
   - 3.1. If $c \leq N$, then the preferred value is $v = 1$
   - 3.2. If $c \geq 2 \cdot (K+1) \cdot N - N$, then $v = 2$
   - 3.3. Otherwise **goto** 1.

Each of those actions, flipping a coin, checking it, and checking the value of c, represents one step in the protocol. Since the processes execute asynchronously, their actions may be arbitrarily interleaved, with interleavings controlled by schedulers. A scheduler may choose from N inputs go_i, one for each process i. Performing go_i basically instructs process i to perform the next step in the protocol. If process i already picked a preferred value in Step 3.1. or in Step 3.2., go_i is simply ignored.

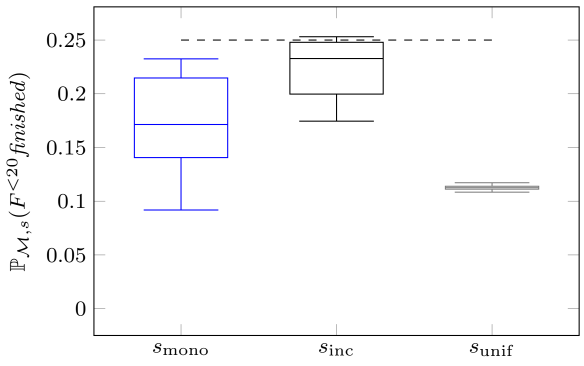
The visible outputs of the system are sets of propositions that hold in the current step. First, the propositions expose the current value of the shared counter, that is, they include $(c = k)$ for a $k \in [0\,..\,2 \cdot (K+1) \cdot N]$. Secondly, they expose values of the local coins, thus the outputs include one $(coin_i = x)$ for each process i, where $x \in \{heads, tails\}$. Additionally, the outputs may include a proposition *finished*, signalling that all processes decided on a preferred value. As generating schedulers for this protocol in a learning-based fashion represents a demanding task, we only consider the case of two asynchronously executing processes by setting $N = 2$ and $K = 2$. Setting either of these constants to larger values significantly increases the number of steps to reach consensus.

Note that information about the current value of local coins is necessary to be able to generate optimal schedulers. Consider the property $\phi = F^{<5}(c = 5)$ and $\max_s \mathbb{P}_{\mathcal{M},s}(\phi)$, which is equal to 0.75, to see why this is the case. Initially, we have $c = 6$ and an optimal scheduler may choose any action, say go_1. After that, we have $coin_1 = heads$ with 0.5 probability and we should perform go_2, because go_1 would increment c. After performing go_2, we have $coin_2 = heads$ with 0.5 probability and we cannot satisfy ϕ anymore. All other traces would satisfy ϕ. Without knowledge about the state of local coins, we would not be able to make sensible choices of inputs. The randomised state machines controlling the processes remain a black box to us, though. Models of their composition are learned.

For the measurements, we optimise $\mathbb{P}_{\mathcal{M},s}(F^{<k} finished)$ for $k \in \{14, 20\}$, thus we try to find schedulings of the two processes which optimise the probability of finishing the protocol in less than 14 steps and 20 steps, respectively. Finishing here means that both processes picked a preferred value.



(a) Reaching *finished* in less than 14 steps



(b) Reaching *finished* in less than 20 steps

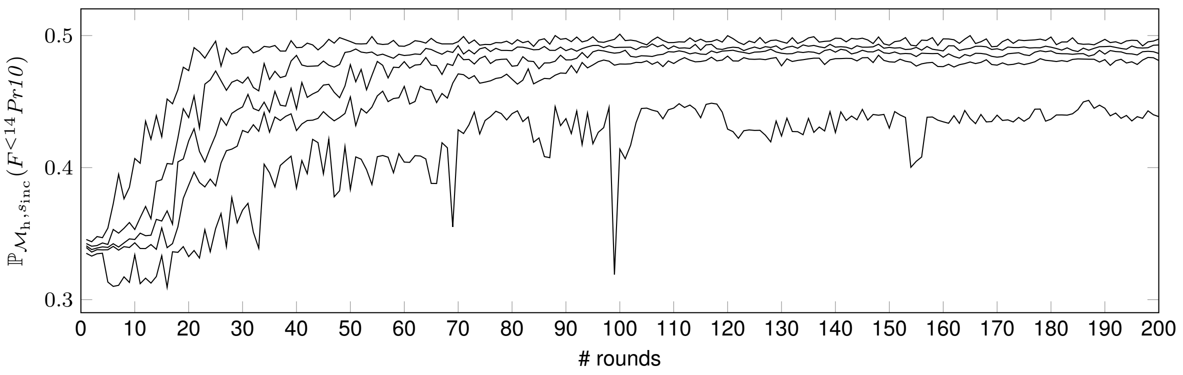
Figure 6.7: Probability estimations of reaching *finished* in the consensus protocol

In the experiments, we applied the configuration given by Table 6.7, setting $maxRounds = 100$, and $n_{\text{batch}} = 250$, and $p_{\text{quit}} = 0.025$. Since we know that states outputting *finished* are absorbing (sink states), we stopped sampling upon seeing the *finished* proposition in an output, as suggested in [198].

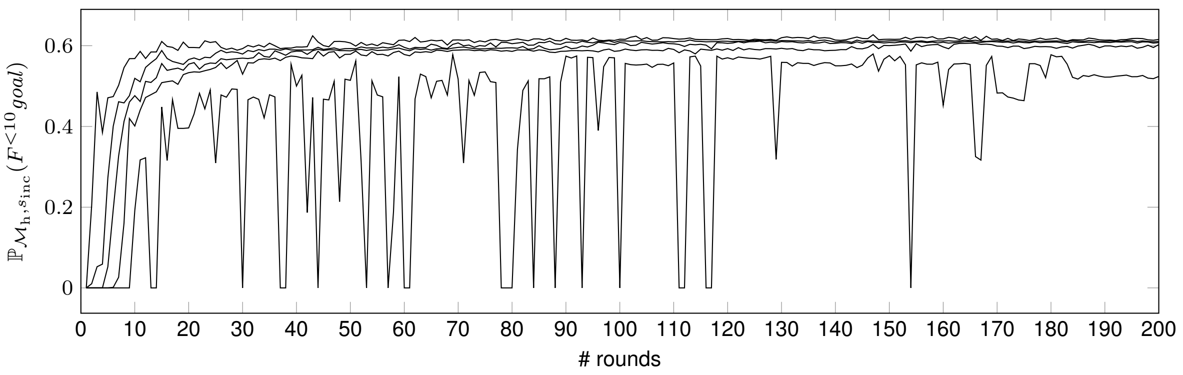
Figure 6.7 shows evaluation results for the incremental and the monolithic approach in comparison to random testing. The box plots corresponding to each of these techniques are labelled s_{mono}, s_{inc}, and s_{unif}, respectively. The dashed line represents the optimal probability as before. In contrast to previous experiments, we see that the monolithic approach may perform worse than random testing. For $k = 14$, there are three measurements, which are exactly zero. However, more than a quarter of the measurement results are near-optimal. For $k = 20$, the number of experiments achieving lower estimations than random testing decreases to two, but none of the generated schedulers is near optimal. This can be explained by considering the minimum number of steps necessary to reach *finished*. We need to execute at least 12 steps to observe *finished*. As a result, it may happen that relevant parts of the system, states reached only after 12 steps, are inaccurately learned. This exemplifies that incremental scheduler generation pays off, because it is able to generate near-optimal schedulers for both values of k. For $k = 14$, three-quarters of the incrementally generated schedulers are near optimal and for $k = 20$, more than one-quarter of the schedulers are near optimal.

It can also be observed that some probability estimations shown in Figure 6.7 are greater than the optimal probability p_{opt}. This may happen, because we apply SMC to compute estimations. Note that all estimations are below $p_{\text{opt}} + \epsilon_{\text{eval}}$, where ϵ_{eval} is the error bound of the applied Chernoff bound [223]. The probability for estimations to exceed $p_{\text{opt}} + \epsilon_{\text{eval}}$ is at most δ_{eval}; see also Section 5.6.

This case study actually highlights a weakness of the convergence check. It assumes that the search will converge to some unique behaviour. The protocol is completely symmetric for both processes, therefore it does not matter which process performs the first step. Hence, there are at least two optimal schedulers which differ only in their initial action. This action is present in each of the 250 traces col-



(a) Statistics for reaching *Pr10* in less than 14 steps (slot machine)



(b) Statistics for reaching *goal* in less than 10 steps (gridworld)

Figure 6.8: Statistics of probability estimations computed in each round of the incremental approach

lected in one round, which presumably include further ambiguous choices. This causes SIMILARSCHED to return **false** in most of the cases. Consequently, we do not discuss results obtained with the convergence check, as it rarely led to early stopping. An approach to counter this problem would be, assuming there is a lexicographic ordering on inputs, to always select the lexicographically minimal input, in case of ambiguous choices.

6.3.7 Convergence Check

We discussed the influence and application of early stopping throughout this section. Now, we want to briefly examine the underlying assumption. The rationale behind the convergence check and early stopping is that scheduler behaviour converges with increasing number of rounds. As a result, fluctuations in the probability estimations produced by schedulers are expected to diminish. Ideally, estimations should increase over time as well. In other words, schedulers should improve. To investigate whether our assumptions hold, we applied the incremental approach and evaluated intermediate schedulers by computing probability estimations in each round of learning.

Figure 6.8 contains graphs showing statistics summarising the computed estimations. The experiment summarised in Figure 6.8a optimises reaching *Pr10* in less than 14 steps with the slot machine. Figure 6.8b shows statistics for reaching *goal* in the gridworld in less than 10 steps. The graphs read as follows: the horizontal axis displays the rounds, and the vertical axis displays the values of the probability estimations. The lines from top to bottom represent the maximum, the third quartile, the median, the first quartile, and the minimum computed from the estimations collected in each round. Like before, these values were calculated from 20 runs.

In both cases we see that fluctuations decrease over time. The interquartile range decreases as well until it becomes relatively stable. Stable estimations are reached at around the 70th round in Figure 6.8a,

Table 6.8: Average runtime of learning and scheduler generation for various properties (all values in seconds)

Case Study & Property	Operation	s_{mono}	s_{inc}	s_{conv}
Slot Machine: $F^{<14}\mathit{Pr10}$	learning	3.4	114.4	64.8
	scheduler generation	3.8	364.6	226.6
MQTT: $F^{<17}crash$	learning	1.7	50.5	75.4
	scheduler generation	2.6	166.4	174.0
TCP: $F^{<17}crash$	learning	21.3	471.7	564.4
	scheduler generation	3.4	256.9	248.3
Gridworld: $F^{<10}goal$	learning	2.1	97.5	96.7
	scheduler generation	2.0	324.6	317.9
Shared Coin: $F^{<20}\mathit{finished}$	learning	7.7	300.9	-
	scheduler generation	3.9	316.8	-

which is the area where convergence was detected – we stopped on average after 71.05 rounds. We see larger fluctuations of the minimal value in Figure 6.8b, but they decline as well. Fluctuations of the minimal value can also be observed after 150 rounds. As a result, we may stop too early in rare cases.

Figure 6.8b also reveals unexpected behaviour. Testing of the gridworld actually required relatively few rounds of learning to achieve good results. In particular, the estimations after the first rounds were larger than expected, because the basis for the first round of learning is formed by only n_{batch} random tests.

6.3.8 Runtime

We simulated previously learned models for the evaluation, thus the simulation cost was very low in our experiments. As a result, the time for learning and reachability checking dominated the computation time. Table 6.8 lists the average runtime of learning and reachability checking for the property with the largest bound of each case study.

It can be seen that the incremental approaches, denoted by s_{inc} and s_{conv}, require considerably more time to complete. Incremental scheduler generation without convergence detection, for instance, takes on average 728.6 seconds for the TCP property $F^{<17}crash$, while the monolithic approach requires only 24.7 seconds. Thus, the better performance with respect to maximising probability estimations comes at the cost of increased runtime for learning and scheduler generation. In a testing scenario with real-world implementations, however, this time overhead may be negligible. If network communication is necessary, for instance, in protocol testing, the simulation time required for interacting with the SUT can be assumed to dominate the overall runtime. This is similar as for our mutation-based test-case generation, which requires more computation time than other approaches, but is more efficient with respect to the simulation time. See Section 4.4 for a discussion on the reduction of required test-case executions.

To contrast simulation runtime with the runtime of learning and scheduler generation, consider the hypothetical, but realistic scenario in which each simulation step takes about 10 milliseconds. In fact, simulation steps may take much longer. A single step in our MQTT case study took up to 600 milliseconds; see Section 3.4. Both approaches, the monolithic and the incremental, require approximately the same number of simulation steps, about $1.7 \cdot 10^6$ for $F^{<17}crash$. In this scenario, the simulation duration would amount to about 4.7 hours, such that the runtime overhead of 703.9 seconds caused by the incremental approach would be low in comparison. Similar observations can be made for other case studies. Since the time spent simulating the SUT can be expected to dominate the computation time, we conclude that the incremental approach is preferable to the monolithic approach in this context.

In Table 6.8, we also see that the convergence detection provides a performance gain for the slot machine, but causes slightly worse runtime for MQTT and TCP. The decreased performance is caused by the fact that convergence often could not be detected within the $maxRounds$ setting used for s_{inc}. Therefore, we increased the $maxRounds$ setting of s_{conv} for MQTT and TCP, as discussed above. However, the goal of convergence detection is not a reduction of runtime. With the convergence detection heuristic, we want to provide a stopping criterion that does not solely rely on an arbitrarily chosen value for the $maxRounds$ parameter.

Finally, we want to discuss the runtime complexity of learning and scheduler generation. The worst-case complexity of IOALERGIA is cubic in the size of the IOFPTA representation of the sampled traces, but the typical behaviour observed in practice is approximately linear [199]. Hence, it is unlikely that learning runtime could be improved, but the scheduler-generation runtime can be improved. Our implementation communicates with PRISM via files, standard input, and standard output. As a result, there is a substantial communication overhead that can be removed via a tighter integration of scheduler generation. PRISM's default technique for scheduler generation, which is called *value iteration*, could also benefit algorithmically from such a tight integration. Since we check reachability with respect to a bound k, it is possible to bound the number of iterations performed by value iteration by k [46]. This leads to a worst-case runtime complexity of $O(k \cdot n^2 \cdot m)$, where n is the number of states of a learned model and m is the number of inputs. The number of inputs m is generally a small constant and we have observed that n^2 is generally smaller than the number of sampling steps required for learning. As a result, we expect the simulation time to generally dominate in non-simulated scenarios.

6.3.9 Discussion

We applied probabilistic black-box reachability checking to various types of models in several configurations. We compared the configurations among each other, to the true optimal testing strategy and also to random testing as a baseline. The results of the performed experiments show (1) that learning-based approaches outperform the baseline and (2) that incremental scheduler learning is able to generate near-optimal schedulers in all cases. In most experiments, the median probability estimation derived with the incremental approach was near optimal, thus it generated near-optimal schedulers reliably. We have also seen that the convergence detection heuristic did not have a negative impact on the accuracy of the incremental approach. However, we are not able to give concrete bounds on the required number of samples to achieve a desired success rate. This is due to the fact that we rely on IOALERGIA, for which convergence in the limit has been shown [199], but stronger guarantees are currently not available.

We generally targeted systems with small state space. An application in practice therefore requires abstraction to ensure that the state space is not prohibitively large. As explained in Section 2.5, this is generally required in learning-based verification and several applications have shown that learning combined with abstraction provides effective means to enable model-based analysis in black-box settings [97, 112, 113], including our work on testing MQTT brokers discussed in Chapter 3 [263].

In addition to that, we have also seen limitations that cannot be solved by abstraction. The differences between estimations and optimal values tend to increase with the step bound k. This is potentially caused by the exponential growth of different traces. This growth also affects the application of the approach for large gridworld examples. Increasing the width of the gridworld also increases the steps required to reach the goal and causes the performance to drop. A possible mitigation would be to identify disabled inputs that are rejected by the SUT if we have such information. This might prevent certain traces from being executed beyond a disabled input. In the original version of IOALERGIA [198], such knowledge facilitates learning, because disabled inputs are assumed to leave the current state unchanged. In the gridworld example, we could consider inputs to be disabled if they cause the robot to move into a wall.

A related issue also affects the case study on the consensus protocol. Changing the number of processes from two to four, increases the minimum number of steps to reach *finished* from 12 to 24. Here, composition actually causes the state space to grow. We could tackle such problems via decomposition. Instead of learning a large monolithic model, several small models could be learned, which would then be composed for the reachability analysis. Recently, Petrenko and Avellaneda have shown that the efficiency of test-based learning can be improved by following such an approach [231]. The authors propose to learn communicating finite-state machines rather than large composite models, as they found that the required amount of testing can be reduced in this way.

6.4 Summary

In this chapter, we presented a learning-based approach for testing stochastic black-box systems with respect to reachability properties. The goal of this approach is to maximise the probability of satisfying such properties, which may, for instance, specify that certain outputs should be produced. More concretely, we developed and implemented an incremental technique that interleaves automata learning, probabilistic model-checking and property-directed testing, to learn near-optimal schedulers for reachability properties of MDPs. The learned schedulers basically serve as online testing-strategies.

To our knowledge, it is the first such approach that is applicable in a purely black-box setting, where only the input interface is known. We evaluated this approach in various configurations and case studies in Section 6.3. The evaluation showed (1) that learning-based testing generally achieves significantly better results than random testing which served as baseline and (2) that near-optimal schedulers can be learned reliably.

6.5 Results and Findings

Our results and findings with respect to relevant research questions introduced in Section 1.6.3 are discussed in this section.

RQ 2.2 When can we stop learning in the presence of uncertain behaviour? We discussed convergence in Section 6.2.2, noting that schedulers learned by our approach converge in the large sample limit to an optimal scheduler. Since there are no PAC guarantees for IoAlergia yet, we cannot determine approximate correctness of models learned from finitely many samples.

In contrast to that, it may be possible to determine online during learning whether we can stop learning. We may stop if (1) the learned scheduler is already sufficiently good, or (2) if the scheduler is unlikely to improve. We addressed the latter by developing a stopping heuristic that checks whether learning has likely converged. We also commented briefly on a scenario, which does not require optimal schedulers. Supppose that a requirement states that the probability of observing a certain output must be lower than p. If an intermediate scheduler achieves to produce this output with an estimated probability $\hat{p}$ larger than p, then we can stop learning. This probability estimate can be computed via SMC, like in the *Evaluate* step, the final step in our process; see Section 6.2.1.

In conclusion, our approach allows for stopping criteria based on schedulers. This is valuable, as the quality of learned models is hard to determine, especially in the absense of PAC learnability guarantees.

RQ 1.1 Are randomised testing techniques a sensible choice for learning-based testing? We applied randomised online testing successfully in the technique presented in this chapter, therefore we consider it a sensible choice. In fact, randomised online testing is commonly applied for systems with uncertain behaviour, for instance, by the tool JTorX [51, 52]. Another example is online testing of real-time systems based on non-deterministic timed automata models [179].

In probabilistic black-box reachability checking, randomly selected inputs provide a way to explore new parts of the SUT. Hence, the rationale behind executing random inputs is similar as for the mutation-based testing approach discussed in Chapter 4.

RQ 1.2 What guarantees can be given if randomised testing is applied? We evaluate learned schedulers via SMC in the last step of the discussed process. This provides us with a probability estimate $\hat{p}$ of satisfying a step-bounded reachability property by controlling the SUT with a learned scheduler. Such an estimate $\hat{p}$ is an **approximate lower bound on the true optimal probability** with respect to some confidence. By applying a Chernoff bound [223] (Equation (5.2)) in the *Evaluate* step, we know with confidence $1 - \delta$ that the true optimal probability must be greater than or equal to $\hat{p} - \epsilon$, where δ and ϵ are the parameters of Equation (5.2).

RQ 1.3 Can learning with randomised conformance testing reliably generate correct system models? We rather tested the SUT to explore its state space than actually testing for conformance. We were able to reliably learn near-optimal schedulers, which is demonstrated by our evaluation in Section 6.3. Since our goal was to learn testing strategies represented by schedulers, the quality of learned system models was not analysed directly.

7 Test-Based Learning of Stochastic Systems

Declaration of Sources

This chapter introduces an active learning technique for MDPs. We presented this technique at FM 2019 [265]. In addition to the conference paper, we developed an extended technical report which is available via `arxiv.org` [266]. This chapter is mainly based on the technical report which includes additional implementation details, proofs as well as an extended evaluation. An evaluation package including the source code of our implementation is available online [261].

7.1 Introduction

Motivation. In Chapter 6, we presented probabilistic black-box reachability checking, a learning-based approach to testing stochastic black-box systems with respect to bounded reachability properties. For that, we applied the state-of-the-art passive learning algorithm IoAlergia [198, 199] in an active setting by sampling through testing based on learned intermediate models. Our evaluation demonstrated that this increases accuracy, as compared to standard random-sampling-based learning. Moreover, with this approach it is possible to reliably produce near-optimal results. A drawback of the technique is that the results are specific to a given reachability property.

This poses the question, whether active test-based learning can improve the accuracy of stochastic models in general. *Compared with passively learned automata, can active test-based learning create MDPs that are closer to the black-box system under learning?* The goal is to answer this question in this chapter, by presenting an L^*-based approach for learning MDPs. The rationale behind this approach is that actively querying the SUL enables to steer the trace generation towards parts of the SUL's state space that have not been thoroughly covered, potentially finding yet unknown aspects of the SUL. In contrast, passive algorithms take a given sample of system traces as input and generate models consistent with the sample. The quality and the comprehensiveness of learned models therefore largely depend on the given sample.

Before going into details, we want to note a few things about our motivating question on whether active learning can produce more accurate MDPs than passive learning. First, we want to emphasise that we consider a *test-based* setting. Like in Chapter 6, we consider a realistic black-box setting, in which we only know an interface to the SUL. Secondly, we want to discuss the actual meaning of *closer* in this context. Similar to Chapter 6, we will present an evaluation based on controlled experiments, in which we know the true model of the SUL. Given such a true model, we apply distance measures, such as the discounted bisimilarity distance [44], to determine how close a learned model is to the true model. Finally, it should be noted why we aim for closeness rather than absolute correctness or probably approximately correct (PAC) learning [276]. IoAlergia learns the correct model in the limit [199]. Currently, it is not possible to make statements about the quality of models learned by IoAlergia from finitely many traces. As a first step, we also strive for guaranteed learning in the limit. For this reason, we empirically investigate how close learned models are to the corresponding true models.

Overview. In this chapter, we present L^*-based learning of MDPs from traces of stochastic black-box systems. For this purpose, we developed two learning algorithms. The first algorithm takes an ideal view assuming perfect knowledge about the exact distribution of system traces. The second algorithm relaxes this assumption, by sampling system traces to estimate their distribution. We refer to the former as *exact learning algorithm* $L^*_{\mathrm{MDP}^e}$ and to the latter as *sampling-based learning algorithm* L^*_{MDP}. We implemented L^*_{MDP} and evaluated it by comparing it to IoAlergia [198, 199]. The experiments presented in this chapter showed favourable performance of L^*_{MDP}. It produced more accurate models than IoAlergia given approximately the same amount of data. Hence, the answer to our motivating question stated above is positive. Active learning can improve accuracy, compared to passive learning.

Apart from the empirical evaluation, we show that the model learned by L^*_{MDP} converges in the limit to an MDP isomorphic to the canonical MDP representing the SUL. To the best of our knowledge, L^*_{MDP} is the first L^*-based learning algorithm for MDPs that can be implemented via testing. Our contributions in this context span the algorithmic development of learning algorithms, their analysis with respect to convergence and the implementation as well as the evaluation of learning algorithms.

Chapter Structure. The rest of this chapter is structured as follows. Section 7.2 discusses observations of MDPs and provides a characterisation of MDPs. Section 7.3 presents the exact learning algorithm $L^*_{\mathrm{MDP}^e}$, while Section 7.4 describes the sampling-based L^*_{MDP}. We analyse L^*_{MDP} with respect to convergence in Section 7.5 and discuss its evaluation in Section 7.6. After that, we provide a summary in Section 7.7 and conclude with remarks on our findings in Section 7.8.

7.2 MDP Observations

In the following, we first introduce different types of observation sequences of MDPs and some related terminology. Then, we formally define the semantics of MDPs in terms of observation sequences.

7.2.1 Sequences of Observations

Let I and O be finite sets of inputs and outputs. Recall that a trace observed during the execution of an MDP is an input-output sequence in $O \times (I \times O)^*$ (see also Section 5.3). We say that a trace t is *observable* if there exists a path ρ with $L(\rho) = t$, thus there is a scheduler s and a length distribution p_l such that $\mathbb{P}^l_{\mathcal{M},s}(t) > 0$. In a deterministic MDP $\mathcal{M}$, each observable trace t uniquely defines a state of $\mathcal{M}$ that is reached by executing t from the initial state q_0. We compute this state by $\delta^*(t) = \delta^*(q_0, t)$ where

$$\delta^*(q, L(q)) = q \text{ and}$$
$$\delta^*(q, o_0 i_1 o_1 \cdots i_{n-1} o_{n-1} i_n o_n) = \Delta(\delta^*(q, o_0 i_1 o_1 \cdots i_{n-1} o_{n-1}), i_n, o_n).$$

If t is not observable, then there is no path ρ with $t = L(\rho)$, denoted by $\delta^*(t) = \bot$. Therefore, we extend Δ by defining $\Delta(\bot, i, o) = \bot$. The last output o_n of a trace $t = o_0 \cdots i_n o_n$ is denoted by $\mathit{last}(t)$.

We use three types of observation sequences with shorthand notations in this chapter:

- *Traces:* abbreviated by $\mathcal{TR} = O \times (I \times O)^*$
- *Test sequences:* abbreviated by $\mathcal{TS} = (O \times I)^*$
- *Continuation sequences:* abbreviated by $\mathcal{CS} = I \times (O \times I)^* = I \times \mathcal{TS}$

These sequence types alternate between inputs and outputs, thus they are related among each other. As mentioned in Section 5.2.2, we extend the sequence notations and the notion of prefixes to sequences containing both inputs and outputs in the context of MDPs. Therefore, we extend these concepts also to $\mathcal{TR}$, $\mathcal{TS}$, and $\mathcal{CS}$. For instance, test sequences and traces are related by $\mathcal{TR} = \mathcal{TS} \cdot O$.

As noted above, an observable trace in $\mathcal{TR}$ leads to a unique state of an MDP $\mathcal{M}$. A test sequence in $s \in \mathcal{TS}$ of length n consists of a trace in $t \in \mathcal{TR}$ with n outputs and an input $i \in I$ with $s = t \cdot i$; thus executing the test sequence $s = t \cdot i$ puts $\mathcal{M}$ into the state reached by t and tests $\mathcal{M}$'s reaction to i. Extending the notion of observability, we say that the test sequence s is observable if t is observable. A continuation sequence $c \in \mathcal{CS}$ begins and ends with an input, i.e. concatenating a trace $t \in \mathcal{TR}$ and c creates a test sequence $t \cdot c$ in $\mathcal{TS}$. Informally, continuation sequences test $\mathcal{M}$'s reaction in response to multiple consecutive inputs.

The following lemma states that an extension of a non-observable traces is also not observable. We will apply this lemma in the context of test-based learning.

Lemma 7.1.
If trace $t \in \mathcal{TR}$ is not observable, then any $t' \in \mathcal{TR}$ such that $t \ll t'$ is not observable as well.

Lemma 7.1 follows directly from Equation (5.1). For a non-observable trace t, we have $\forall s, p_l : \mathbb{P}^l_{\mathcal{M},s}(t) = 0$ and extending t to create t' only adds further factors to Equation (5.1). The same property holds for test sequences as well.

7.2.2 Semantics of MDPs

We can interpret an MDP as a function $M : \mathcal{TS} \to \mathit{Dist}(O) \cup \{\bot\}$, mapping test sequences s to output distributions or undefined behaviour for non-observable s. This follows the interpretation of Mealy machines as functions from input sequences to outputs [258]. Likewise, we will define which functions M capture the semantics of MDPs by adapting the Myhill-Nerode theorem on regular languages [216]. In the remainder of this chapter, we denote the set of test sequences s where $M(s) \neq \bot$ as defined domain $dd(M)$ of M.

Definition 7.1 (MDP Semantics).
The semantics of an MDP $\langle Q, I, O, q_0, \delta, L\rangle$, is a function M, defined for $i \in I$, $o \in O$, and $t \in \mathcal{TR}$ as follows:

$$\begin{aligned}
M(\epsilon)(L(q_0)) &= 1 \\
M(t \cdot i) &= \bot \text{ if } \delta^*(t) = \bot \\
M(t \cdot i)(o) &= p \text{ if } \delta^*(t) \neq \bot \land \delta(\delta^*(t), i)(q) = p > 0 \land L(q) = o \\
M(t \cdot i)(o) &= 0 \text{ if } \delta^*(t) \neq \bot \land \nexists\, q : \delta(\delta^*(t), i)(q) = p > 0 \land L(q) = o
\end{aligned}$$

Definition 7.2 (M-Equivalence of Traces).
Two traces $t_1, t_2 \in \mathcal{TR}$ are equivalent with respect to $M : \mathcal{TS} \to \mathit{Dist}(O) \cup \{\bot\}$, denoted $t_1 \equiv_M t_2$, iff

- *$last(t_1) = last(t_2)$ and*
- *it holds for all continuations $v \in \mathcal{CS}$ that $M(t_1 \cdot v) = M(t_2 \cdot v)$.*

A function M defines an equivalence relation on traces, like the Myhill-Nerode equivalence for formal languages [216]. Two traces are M-equivalent if they end in the same output and if their behaviour in response to future inputs is the same. Two traces leading to the same MDP state are in the same equivalence class of $\equiv_M$, analogously to the adapted Myhill-Nerode equivalence for Mealy machines [258].

We can now state which functions characterise MDPs, as an adaptation of the Myhill-Nerode theorem for regular languages [216], like for Mealy machines [258].

Theorem 7.1 (Characterisation).
A function $M : \mathcal{TS} \to Dist(O) \cup \{\bot\}$ represents the semantics of an MDP iff

1. *$\equiv_M$ has finite index,* *... finite number of states*
2. *$M(\epsilon) = \mathbf{1}_{\{o\}}$ for an $o \in O$,* *... initial output*
3. *$dd(M)$ is prefix-closed, and*
4. *$\forall t \in \mathcal{TR}$: either $\forall i \in I : M(t \cdot i) \neq \bot$ or $\forall i \in I : M(t \cdot i) = \bot$* *... input enabledness*

Proof. Direction ⇒: first we show that the semantics M of an MDP $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$ fulfils the four conditions of Theorem 7.1. Let $t \in \mathcal{TR}$ be an observable trace, then we have for $i \in I, o \in O$: $M(t \cdot i)(o) = \delta(q', i)(q) = p$, where $q' = \delta^*(t)$, if $p > 0$ and $L(q) = o$. As $\mathcal{M}$ contains finitely many states q', $\delta(q', i)$ and therefore also $M(t \cdot i)$ takes only finitely many values. M-equivalence of traces t_i depends on the outcomes of M and on their last outputs $last(t_i)$. Since the set of possible outputs is finite like the possible outcomes of M, M-equivalence defines finitely many equivalence classes for observable traces. For non-observable $t \in \mathcal{TR}$ we have $\delta^*(t) = \bot$ which implies $M(t \cdot i) = \bot$. As a consequence of Lemma 7.1, we also have $M(t \cdot c) = \bot$ for any $c \in \mathcal{CS}$. Hence, non-observable traces are equivalent with respect to M if they end in the same output, therefore M defines finitely many equivalence classes for non-observable traces. In summary, $\equiv_M$ has finite index, which fulfils the first condition.

According to Definition 7.1, it holds that $M(\epsilon)(L(q_0)) = 1$, thus the second condition is fulfilled.

Prefix-closedness of the defined domain $dd(M)$ of M follows from Lemma 7.1. Any extension of a non-observable test sequence is also non-observable, thus M fulfils the third condition.

For the fourth condition, we again distinguish two cases. If t is a non-observable trace, i.e. $\delta^*(t) = \bot$, then $M(t \cdot i) = \bot$ for all $i \in I$ according to Definition 7.1, which fulfils the second sub-condition. For observable t, the distribution $M(t \cdot i)$ depends on $\delta(\delta^*(t), i)$, which is defined for all i due to input enabledness of $\mathcal{M}$, satisfying the first subcondition.

The semantics of an MDP satisfies all four conditions listed in Theorem 7.1.

Direction ⇐: from an M satisfying the conditions given in Theorem 7.1, we can construct an MDP $\mathcal{M}_c = \langle Q, I, O, q_0, \delta, L\rangle$ by:

- $Q = (\mathcal{TR}/\equiv_M) \setminus \{[t] \mid t \in \mathcal{TR}, \exists i \in I : M(t \cdot i) = \bot\}$
- $q_0 = [o_0]$, where $o_0 \in O$ such that $M(\epsilon) = \mathbf{1}_{\{o_0\}}$
- $L([s \cdot o]) = o$ where $s \in \mathcal{TS}$ and $o \in O$, by Definition 7.2 all traces in the same equivalence class end with the same output
- for $[t] \in Q$:
  $\delta([t], i)([t \cdot i \cdot o]) = M(t \cdot i)(o)$, defined by fourth condition of Theorem 7.1

Each equivalence class of $\equiv_M$ gives rise to exactly one state in Q, except for the equivalence classes of non-observable traces. □

The MDP $\mathcal{M}_c$ in the construction above is minimal with respect to the number of states and unique up to isomorphism. Therefore, we refer to an MDP, constructed in this way, as canonical MDP $can(M)$ for MDP semantics M. $\mathcal{M}_c$ is minimal, because any other MDP $\mathcal{M}'$ with a lower number of states necessarily contains a state that is reached by traces from different M-equivalence classes. Consequently, $\mathcal{M}'$ cannot be consistent with the semantics M.

Viewing MDPs as reactive systems, we consider two MDPs to be equivalent, if we make the same observations on both.

Definition 7.3 (Output-Distribution Equivalence).
MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$ over I and O with semantics M_1 and M_2 are output-distribution equivalent, denoted $\mathcal{M}_1 \equiv_{\mathrm{od}} \mathcal{M}_2$, iff

$$\forall s \in \mathcal{TS} : M_1(s) = M_2(s)$$

7.3 Exact Learning of MDPs

This section presents $L^*_{\mathrm{MDP}^e}$, an exact active learning algorithm for MDPs. $L^*_{\mathrm{MDP}^e}$ serves as basis for the sampling-based learning algorithm presented in Section 7.4. In contrast to sampling, $L^*_{\mathrm{MDP}^e}$ assumes the existence of a teacher with perfect knowledge about the SUL that is able to answer two types of queries: *output distribution* queries and *equivalence* queries. The former asks for the exact distribution of outputs following the execution of a test sequence on the SUL. The latter takes a hypothesis MDP as input and responds either with *yes* iff the hypothesis is observationally equivalent to the SUL or with a counterexample to equivalence. A counterexample is a test sequence leading to different output distributions in hypothesis and SUL.

7.3.1 Queries

Before discussing learning, we formally define the queries available to the learner that focus on the observable behaviour of MDPs. Assume that we want to learn a model of a black-box deterministic MDP $\mathcal{M}$, with semantics M. Output distribution queries ($\mathbf{odq}$) and equivalence queries ($\mathbf{eq}$) are then defined as follows:

- *output distribution query* ($\mathbf{odq}$): an $\mathbf{odq}(s)$ returns $M(s)$ for input $s \in \mathcal{TS}$.
- *equivalence query* ($\mathbf{eq}$): an $\mathbf{eq}$ query takes a hypothesis MDP $\mathcal{H}$ with semantics H as input and returns *yes* if $\mathcal{H} \equiv_{\mathrm{od}} \mathcal{M}$; otherwise it returns an $s \in \mathcal{TS}$ such that $H(s) \neq M(s)$ and $M(s) \neq \bot$.

Lemma 7.2 (Counterexample Observability).
For any counterexample s to $\mathcal{H} \equiv_{\mathrm{od}} \mathcal{M}$ with $M(s) = \bot$, there exists a prefix s' of s with $H(s') \neq M(s')$ and $M(s') \neq \bot$, thus s' is also a counterexample but observable on the SUL with semantics M. Hence, we can restrict potential counterexamples to be observable test sequences.

Proof. Since s is a counterexample and $M(s) = \bot$, we have $H(s) \neq \bot$. Let s'' be the the longest prefix of s such that $M(s'') = \bot$, thus s'' is of the form $s'' = s' \cdot o \cdot i$ with $M(s')(o) = 0$. Due to prefix-closedness of $dd(H)$, $H(s) \neq \bot$ implies $H(s'') \neq \bot$, therefore $H(s')(o) > 0$. Hence, s' with $M(s') \neq \bot$ is also a counterexample because $H(s')(o) \neq M(s')(o) \Rightarrow H(s') \neq M(s')$. □

7.3.2 Observation Tables

Like Angluin's L^* [37], we store information in observation table triples $\langle S, E, T \rangle$, where:

- $S \subseteq \mathcal{TR}$ is a prefix-closed set of traces, initialised to $\{o_0\}$, a singleton set containing the trace consisting of the initial output o_0 of the SUL, given by $\mathbf{odq}(\epsilon)(o_0) = 1$,

Table 7.1: An observation table created during learning of the faulty coffee machine (Figure 5.1)

		but	coin
S	init · coin	$\{$init $\mapsto 1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep	$\{$coffee $\mapsto 0.9$, init $\mapsto 0.1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep · but · coffee	$\{$init $\mapsto 1\}$	$\{$beep $\mapsto 1\}$
$Lt(S)$	init · but · init	$\{$init $\mapsto 1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep · but · init	$\{$init $\mapsto 1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep · coin · beep	$\{$coffee $\mapsto 0.9$, init $\mapsto 0.1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep · but · coffee· but · init	$\{$init $\mapsto 1\}$	$\{$beep $\mapsto 1\}$
	init · coin · beep · but · coffee· coin · beep	$\{$coffee $\mapsto 0.9$, init $\mapsto 0.1\}$	$\{$beep $\mapsto 1\}$

- $E \subseteq \mathcal{CS}$ is a suffix-closed set of continuation sequences, initialised to I,
- $T : (S \cup Lt(S)) \cdot E \to Dist(O) \cup \{\bot\}$ is a mapping from test sequences to output distributions or to $\bot$, denoting undefined behaviour. This mapping basically stores a finite subset of M. The set $Lt(S) \subseteq S \cdot I \cdot O$ is given by $Lt(S) = \{s \cdot i \cdot o \mid s \in S, i \in I, o \in O, \mathbf{odq}(s \cdot i)(o) > 0\}$.

We can view an observation table as a two-dimensional array with rows labelled by traces in $S \cup Lt(S)$ and columns labelled by E. We refer to traces in S as short traces and to their extensions in $Lt(S)$ as long traces. An extension $s \cdot i \cdot o$ of a short trace s is in $Lt(S)$ if $s \cdot i \cdot o$ is observable. Analogously to traces, we refer to rows labelled by S as short rows and we refer to rows labelled by $Lt(S)$ as long rows. The table cells store the mapping defined by T. To represent rows labelled by traces s we use functions $row(s) : E \to Dist(O) \cup \{\bot\}$ for $s \in S \cup Lt(S)$ with $row(s)(e) = T(s \cdot e)$. Equivalence of rows labelled by traces s_1, s_2, denoted $\mathbf{eqRow}_E(s_1, s_2)$, holds iff $row(s_1) = row(s_2) \wedge last(s_1) = last(s_2)$. It approximates M-equivalence $s_1 \equiv_M s_2$ by considering only continuations in E, hence $s_1 \equiv_M s_2$ implies $\mathbf{eqRow}_E(s_1, s_2)$. The observation table content defines the structure of hypothesis MDPs based on the following principle: we create one state per equivalence class of $S/\mathbf{eqRow}_E$, thus we identify states with traces in S reaching them and we distinguish states by their future behaviour in response to sequences in E. The long traces $Lt(S)$ serve to define transitions. As discussed in Section 2.2.2, this is a common approach to hypothesis construction in active automata learning [258]. Transition probabilities are given by the distributions in the mapping T.

Example 7.1 (Observation Table for Coffee Machine). Table 7.1 shows an observation table created during learning of the coffee machine shown in Figure 5.1. The hypothesis derived from that observable table is already equivalent to the true model. The set S includes a trace for each state of the coffee machine. Note that these traces are pairwise inequivalent with respect to $\mathbf{eqRow}_E$, where $E = I = \{$but, coin$\}$. As noted above, the traces labelled by $Lt(S)$ define transitions in hypotheses. For instance, the row labelled by init · but · init gives rise to the self-loop transition in the initial state with the input but and probability 1. This is the case, because the rows labelled by init · but · init and init are equivalent, where init $\in S$ corresponds to the initial state of the hypothesis.

Definition 7.4 (Closedness).
An observation table $\langle S, E, T\rangle$ is closed if for all $l \in Lt(S)$ there is an $s \in S$ such that $\mathbf{eqRow}_E(l, s)$.

Definition 7.5 (Consistency).
An observation table $\langle S, E, T\rangle$ is consistent if for all $s_1, s_2 \in S, i \in I, o \in O$ such that $\mathbf{eqRow}_E(s_1, s_2)$ *it holds either that (1)*[1] $T(s_1 \cdot i)(o) = 0 \wedge T(s_2 \cdot i)(o) = 0$ *or (2)* $\mathbf{eqRow}_E(s_1 \cdot i \cdot o, s_2 \cdot i \cdot o)$.

[1] Note that $s_1 \in S$ implies that $T(s_1 \cdot i) \neq \bot$ such that $T(s_2 \cdot i)(o) = 0$ follows from $\mathbf{eqRow}_E(s_1, s_2)$ and $T(s_1 \cdot i)(o) = 0$.

Algorithm 7.1 Function for making an observation table closed and consistent

```
1: function MAKECLOSEDANDCONSISTENT($\langle S, E, T\rangle$)
2:     if $\langle S, E, T\rangle$ is not closed then
3:         $l \leftarrow l' \in Lt(S)$ such that $\forall s \in S : row(s) \neq row(l') \vee last(s) \neq last(l')$
4:         $S \leftarrow S \cup \{l\}$
5:         return $\langle S, E, T\rangle$
6:     else if $\langle S, E, T\rangle$ is not consistent then
7:         for all $s_1, s_2 \in S$ such that $\mathbf{eqRow}_E(s_1, s_2)$ do
8:             for all $i \in I, o \in O$ do
9:                 if $T(s_1 \cdot i)(o) > 0$ and $\neg\mathbf{eqRow}_E(s_1 \cdot i \cdot o, s_2 \cdot i \cdot o)$ then
10:                    $e \leftarrow e' \in E$ such that $T(s_1 \cdot i \cdot o \cdot e') \neq T(s_2 \cdot i \cdot o \cdot e')$
11:                    $E \leftarrow E \cup \{i \cdot o \cdot e\}$
12:                    return $\langle S, E, T\rangle$
13:                end if
14:            end for
15:        end for
16:    end if
17:    return $\langle S, E, T\rangle$        ▷ reached if already closed and consistent
18: end function
```

Closedness and consistency are required to derive well-formed hypotheses, analogously to L^* [37]. We require closedness to create transitions for all inputs in all states and we require consistency to be able to derive deterministic hypotheses. During learning, we apply Algorithm 7.1 repeatedly to establish closedness and consistency of observation tables. The algorithm adds a new short trace if the table is not closed and adds a new column if the table is not consistent.

We derive a hypothesis $\mathcal{H} = \langle Q_\mathrm{h}, I, O, q_{0\mathrm{h}}, \delta_\mathrm{h}, L_\mathrm{h}\rangle$ from a closed and consistent observation table $\langle S, E, T\rangle$, denoted $\mathcal{H} = \mathrm{hyp}(S, E, T)$, as follows:

- $Q_\mathrm{h} = \{\langle last(s), row(s)\rangle \mid s \in S\}$
- $q_{0\mathrm{h}} = \langle o_0, row(o_0)\rangle$, $o_0 \in S$ is the trace consisting of the initial SUL output
- for $s \in S$, $i \in I$ and $o \in O$:
  $\delta_\mathrm{h}(\langle last(s), row(s)\rangle, i)(\langle o, row(s \cdot i \cdot o)\rangle) = p$ if $T(s \cdot i)(o) = p > 0$ and 0 otherwise
- for $s \in S$: $L_\mathrm{h}(\langle last(s), row(s)\rangle) = last(s)$

7.3.3 Learning Algorithm

Algorithm 7.2 implements $L^*_{\mathrm{MDP}^e}$ using queries $\mathbf{odq}$ and $\mathbf{eq}$. First, the algorithm initialises the observation table and fills the table cells with output distribution queries (lines 1 to 3). The main loop in lines 4 to 18 makes the observation table closed and consistent, derives a hypothesis $\mathcal{H}$ and performs an equivalence query $\mathbf{eq}(\mathcal{H})$. If $\mathbf{eq}(\mathcal{H})$ returns a counterexample *cex*, we add all its prefix traces as short traces to S, otherwise Algorithm 7.2 returns the final hypothesis, as it is output-distribution equivalent to the SUL. Whenever the observation table contains empty cells, the FILL procedure assigns values to these cells via $\mathbf{odq}$.

Note that we use the classic L^*-style counterexample processing, because other techniques to process counterexamples are hard to apply efficiently in a sampling-based setting. It would be possible to adapt Rivest and Schapire's counterexample processing [240] in the exact setting considered in this section, but the main purpose of introducing $L^*_{\mathrm{MDP}^e}$ is to provide a basis for the sampling-based L^*_{MDP}.

Algorithm 7.2 The main algorithm implementing the exact $L^*_{\mathrm{MDP}^e}$

```
Input: $I$, exact teacher capable of answering $\mathbf{odq}$ and $\mathbf{eq}$
Output: learned model $\mathcal{H}$ (final hypothesis)
 1: $o_0 \leftarrow o$ such that $\mathbf{odq}(\epsilon)(o) = 1$
 2: $S \leftarrow \{o_0\}$, $E \leftarrow I$
 3: FILL($S, E, T$)
 4: repeat
 5:     while $\langle S, E, T\rangle$ not closed or not consistent do
 6:         $\langle S, E, T\rangle \leftarrow$ MAKECLOSEDANDCONSISTENT($\langle S, E, T\rangle$)
 7:         FILL($S, E, T$)
 8:     end while
 9:     $\mathcal{H} \leftarrow \mathrm{hyp}(S, E, T)$
10:     $eqResult \leftarrow \mathbf{eq}(\mathcal{H})$
11:     if $eqResult \neq yes$ then
12:         $cex \leftarrow eqResult$
13:         for all $(t \cdot i) \in prefixes(cex)$ with $i \in I$ do
14:             $S \leftarrow S \cup \{t\}$
15:         end for
16:         FILL($S, E, T$)
17:     end if
18: until $eqResult = yes$
19: return $\mathrm{hyp}(S, E, T)$

20: procedure FILL($S, E, T$)
21:     for all $s \in S \cup Lt(S), e \in E$ do
22:         if $T(s \cdot e)$ undefined then          ▷ we have no information about $T(s \cdot e)$ yet
23:             $T(s \cdot e) \leftarrow \mathbf{odq}(s \cdot e)$
24:         end if
25:     end for
26: end procedure
```

7.3.4 Correctness & Termination

In the following, we will show that $L^*_{\mathrm{MDP}^e}$ terminates and learns correct models. We consider learned models correct if they are output-distribution equivalent to the SUL. Like Angluin [37], we will show that derived hypotheses are consistent with queried information and that they are minimal with respect to the number of states. For the remainder of this section, let M be the semantics of the MDP underlying the SUL and let $\mathcal{M} = can(M)$ be the corresponding canonical MDP and let $\mathcal{H} = \langle Q, I, O, q_0, \delta, L\rangle$ denote hypotheses. The first two lemmas relate to observability of traces.

Lemma 7.3.
For all $s \in \mathcal{TS}, o \in O, e \in \mathcal{CS} : M(s)(o) = 0 \Rightarrow M(s \cdot o \cdot e) = \bot$.

Proof. Let δ_M be the probabilistic transition relation of $\mathcal{M}$. $M(s)(o) = 0$ with $s = t \cdot i$ for $i \in I$ implies that there is no state labelled o reachable by executing i in the state $\delta^*_M(t)$ (Definition 7.1), thus $\delta^*_M(t \cdot i \cdot o) = \delta^*_M(s \cdot o) = \bot$. By Definition 7.1, $M(s \cdot o \cdot i') = \bot$ for any i'. Due to prefix-closedness of $dd(M)$, we have $M(s \cdot o \cdot e) = \bot$ for all $e \in \mathcal{CS}$. □

Lemma 7.4.
Let $\langle S, E, T\rangle$ *be a closed and consistent observation table. Then for* $s \in S$ *and* $s \cdot i \cdot o \in S \cup Lt(S)$ *we have* $T(s \cdot i)(o) > 0$.

Proof. The lemma states that traces labelling rows are observable. Algorithm 7.2 adds elements to S and consequently to $Lt(S)$ in two cases: (1) if an equivalence query returns a counterexample and (2) to make observation tables closed.

Case 1. Counterexamples $c \in \mathcal{TS}$ returned by equivalence queries $\mathbf{eq}(\mathcal{H})$ satisfy $M(c) \neq \bot$ (see also Lemma 7.2). In Line 14 of Algorithm 7.2, we add t_p to S for each $t_p \cdot i_p \in \mathit{prefixes}(c)$. Due to prefix-closedness of $dd(M)$, $M(t_p \cdot i_p) \neq \bot$ for all $t_p \cdot i_p \in \mathit{prefixes}(c)$, and therefore $M(s \cdot i)(o) = T(s \cdot i)(o) > 0$ for each added trace t_p of the form $t_p = s \cdot i \cdot o$ with $i \in I$ and $o \in O$. Due to its definition, the set $Lt(S)$ is implicitly extended by all observable extensions of added t_p. By this definition, $Lt(S)$ contains only traces $t = s \cdot i \cdot o$ such that $T(s \cdot i)(o) > 0$.

Case 2. If an observation table is not closed, we add traces from $Lt(S)$ to S. As noted above, all traces $t = s \cdot i \cdot o$ in $Lt(S)$ satisfy $T(s \cdot i)(o) > 0$. Consequently, all traces added to S satisfy this property as well. □

Theorem 7.2 (Consistency and Minimality).
Let $\langle S, E, T\rangle$ be a closed and consistent observation table and let $\mathcal{H} = \mathrm{hyp}(S, E, T)$ be a hypothesis derived from that table with semantics H. Then $\mathcal{H}$ is consistent with T, that is, $\forall s \in (S \cup Lt(s)) \cdot E : T(s) = H(s)$, and any other MDP consistent with T but inequivalent to $\mathcal{H}$ must have more states.

The following four lemmas are necessary to prove Theorem 7.2.

Lemma 7.5.
Let $\langle S, E, T\rangle$ be a closed and consistent observation table. For $\mathcal{H} = \mathrm{hyp}(S, E, T)$ and every $s \in S \cup Lt(S)$, we have $\delta^*(q_0, s) = \langle \mathit{last}(s), \mathit{row}(s)\rangle$.

Proof. Similarly to Angluin [37], we prove this by induction on the length k of s. In Section 5.2.2, we defined the length of a trace as its number of input-output pairs. Hence, for the base case of $k = 0$, we consider the trace s consisting of only the initial output o, that is, $s = o$. In this case, we have

$$\delta^*(q_0, o) = \delta^*(\langle o, \mathit{row}(o)\rangle, o) = \langle o, \mathit{row}(o)\rangle.$$

Assume that for every $s \in S \cup Lt(S)$ of length at most k, $\delta^*(q_0, s) = \langle \mathit{last}(s), \mathit{row}(s)\rangle$. Let $t \in S \cup Lt(S)$ be a trace of length $k+1$. Such a t is of the form $t = s \cdot i \cdot o_t$, with $s \in \mathcal{TR}, i \in I, o_t \in O$. If $t \in Lt(S)$ then s must be in S, and if $t \in S$, then also $s \in S$ because S is prefix-closed.

$$\begin{aligned}
\delta^*(q_0, s \cdot i \cdot o_t) &= \Delta(\delta^*(q_0, s), i, o_t) && \text{(definition of } \delta^*) \\
&= \Delta(\langle \mathit{last}(s), \mathit{row}(s)\rangle, i, o_t) && \text{(by induction hypothesis)} \\
&= \langle o_t, \mathit{row}(s \cdot i \cdot o_t)\rangle && \text{(definition of } \Delta) \\
&\text{if } \delta(\langle \mathit{last}(s), \mathit{row}(s)\rangle, i)(\langle o_t, \mathit{row}(s \cdot i \cdot o_t)\rangle) > 0 \\
&\text{and } L(\langle o_t, \mathit{row}(s \cdot i \cdot o_t)\rangle) = o_t
\end{aligned}$$

$$\begin{aligned}
&\delta(\langle \mathit{last}(s), \mathit{row}(s)\rangle, i)(\langle o_t, \mathit{row}(s \cdot i \cdot o_t)\rangle) > 0 \\
&\qquad \Leftrightarrow T(s \cdot i)(o_t) > 0 && \text{(construction of } \delta) \\
&\qquad \Leftrightarrow \mathbf{true} && \text{(Lemma 7.4)} \\
&L(\langle o_t, \mathit{row}(s \cdot i \cdot o_t)\rangle) = o_t \\
&\qquad \Leftrightarrow \mathbf{true} && \text{(construction of } L)
\end{aligned}$$

□

Lemma 7.6.
Let (S, E, T) be a closed and consistent observation table. Then $\mathrm{hyp}(S, E, T)$ is consistent with T, i.e. for every $s \in S \cup Lt(S)$ and $e \in E$ we have $T(s \cdot e) = H(s \cdot e)$.

Proof. We will prove this by induction on k, the number of inputs in e. As induction hypothesis, we assume that $T(s \cdot e) = H(s \cdot e)$ for all $s \in S \cup Lt(S)$ and $e \in E$ containing at most k inputs. For the base case with $k = 1$, we consider e consisting of a single input, that is, $e \in I$. From Definition 7.1 we can derive that $H(s \cdot i) \neq \bot$ if $\delta^*(s) \neq \bot$, which holds for $s \in S \cup Lt(S)$. Then we have:

$$\begin{aligned}
H(s \cdot i)(o) &= \delta(\delta^*(s), i)(q) \text{ with } L(q) = o \\
&= \delta(\langle \mathit{last}(s), \mathit{row}(s) \rangle, i)(q) \text{ with } L(q) = o && \text{(Lemma 7.5)} \\
&= \delta(\langle \mathit{last}(s), \mathit{row}(s) \rangle, i)(\langle o, \mathit{row}(s \cdot i \cdot o) \rangle) && \text{(hypothesis construction)} \\
&= T(s \cdot i)(o) && \text{(hypothesis construction)}
\end{aligned}$$

For the induction step, let $e \in E$ be such that it contains $k+1$ inputs, thus it is of the form $e = i \cdot o \cdot e_k$ for $i \in I$, $o \in O$, and due to suffix-closedness of E, $e_k \in E$. Note that e_k contains k inputs. We have to show that $T(s \cdot e) = H(s \cdot e)$ for $s \in S \cup Lt(S)$. Let $s' \in S$ such that $\mathbf{eqRow}_E(s, s')$, which exists due to observation table closedness. Traces s and s' lead to the same hypothesis state because:

$$\begin{aligned}
\delta^*(q_0, s) &= \langle \mathit{last}(s), \mathit{row}(s) \rangle && \text{(Lemma 7.5)} \\
&= \langle \mathit{last}(s'), \mathit{row}(s') \rangle && (\mathbf{eqRow}_E(s, s')) \\
&= \delta^*(q_0, s') && \text{(Lemma 7.5)}
\end{aligned}$$

Thus, s and s' are H-equivalent and therefore $H(s \cdot e) = H(s' \cdot e)$. Due to $\mathbf{eqRow}_E(s, s')$, $T(s \cdot e) = T(s' \cdot e)$ and in combination:

$$T(s \cdot e) = H(s \cdot e) \Leftrightarrow T(s' \cdot e) = H(s' \cdot e) \Leftrightarrow T(s' \cdot i \cdot o \cdot e_k) = H(s' \cdot i \cdot o \cdot e_k)$$

We will now show that $T(s' \cdot i \cdot o \cdot e_k) = H(s' \cdot i \cdot o \cdot e_k)$ holds by considering two cases.

Case 1. Suppose that $s' \cdot i \cdot o \in S \cup Lt(S)$. Then, $T(s' \cdot i \cdot o \cdot e_k) = H(s' \cdot i \cdot o \cdot e_k)$ holds by the induction hypothesis, as e_k contains k inputs.

Case 2. Suppose that $s' \cdot i \cdot o \notin S \cup Lt(S)$. Due to $s' \in S$ and the definition of $Lt(S)$, we have

$$\mathbf{odq}(s' \cdot i)(o) = M(s' \cdot i)(o) = 0.$$

By Lemma 7.3, it follows that $M(s' \cdot i \cdot o \cdot e) = \bot$ for any continuation $e \in \mathcal{CS}$. Since the observation table is filled via $\mathbf{odq}$ we have

$$T(s' \cdot i \cdot o \cdot e_k) = \mathbf{odq}(s' \cdot i \cdot o \cdot e_k) = M(s' \cdot i \cdot o \cdot e_k) = \bot.$$

By the induction base, we have $H(s' \cdot i) = T(s' \cdot i)$ for $i \in I$, thus $H(s' \cdot i)(o) = T(s' \cdot i)(o)$, for which we know $T(s' \cdot i)(o) = 0$, because $s' \cdot i \cdot o \notin S \cup Lt(S)$. Combined with Lemma 7.3, we can conclude that $H(s' \cdot i \cdot o \cdot e_k) = \bot$.

In both cases, it holds that $H(s' \cdot i \cdot o \cdot e_k) = T(s' \cdot i \cdot o \cdot e_k)$ which is equivalent to $H(s \cdot e) = T(s \cdot e)$. $\square$

With Lemma 7.6, we have shown consistency between derived hypotheses and the queried information stored in T. Now, we show that hypotheses are minimal with respect to the number of states.

Lemma 7.7.
Let $\langle S, E, T \rangle$ be a closed and consistent observation table and let n be the number of different values of $\langle \mathit{last}(s), \mathit{row}(s) \rangle$, i.e. the hypothesis $\mathrm{hyp}(S, E, T)$ *has n states due to its construction. Any MDP consistent with T must have at least n states.*

Proof. Let $\mathcal{M}' = \langle Q', I, O, q_0', \delta', L' \rangle$ with semantics M' be an MDP that is consistent with T. Let $s_1, s_2 \in S$ be such that $\neg\mathbf{eqRow}_E(s_1, s_2)$, then (1) $\mathit{last}(s_1) \neq \mathit{last}(s_2)$ or (2) $\mathit{row}(s_1) \neq \mathit{row}(s_2)$. If $\mathit{last}(s_1) \neq \mathit{last}(s_2)$, then s_1 and s_2 cannot reach the same state in $\mathcal{M}'$, because the states reached by s_1 and s_2 need to be labelled differently. If $\mathit{row}(s_1) \neq \mathit{row}(s_2)$, then there exists an $e \in E$ such that

$T(s_1 \cdot e) \neq T(s_2 \cdot e)$. Since $\mathcal{M}'$ is consistent with T, it holds also that $M'(s_1 \cdot e) \neq M'(s_2 \cdot e)$. In this case, s_1 and s_2 cannot reach the same state as well, as the observed future behaviour is different.

Consequently, $\mathcal{M}'$ has at least n states, as there must exist at least one state for each value of $\langle \mathit{last}(s), \mathit{row}(s)\rangle$. Two traces s_1 and s_2 satisfying $\langle \mathit{last}(s_1), \mathit{row}(s_1)\rangle \neq \langle \mathit{last}(s_2), \mathit{row}(s_2)\rangle$ cannot reach the same state. □

Lemma 7.8.
Let $\langle S, E, T\rangle$ be a closed and consistent observation table and $\mathcal{H} = \mathrm{hyp}(S, E, T)$ be a hypothesis with n states derived from it. Any other MDP $\mathcal{M}' = \langle Q', I, O, q'_0, \delta', L'\rangle$ with semantics M' consistent with T, initial output $L'(q'_0) = L(q_0)$, and with n or fewer states is isomorphic to $\mathcal{H}$.

Proof. From Lemma 7.7, it follows that $\mathcal{M}'$ has at least n states, therefore we examine $\mathcal{M}'$ with exactly n states. For each state of $\mathcal{H}$, i.e. for each unique row value $\langle \mathit{last}(s), \mathit{row}(s)\rangle$ labelled by some $s \in S$, exists a unique state in Q'. Let ϕ be a mapping from short traces to Q' given by $\phi(\langle \mathit{last}(s), \mathit{row}(s)\rangle) = \delta'^*(q'_0, s)$ for s in S. This mapping is bijective and we will now show that it maps q_0 to q'_0, that it preserves the probabilistic transition relation and that it preserves labelling.

First, we start with the initial state and show $\phi(q_0) = q'_0$:

$$\begin{aligned}\phi(q_0) &= \phi(\langle o, \mathit{row}(o)\rangle) \text{ where } o \text{ is the initial output of the SUL}\\ &= \delta'^*(q'_0, o)\\ &= q'_0 && \text{(definition of } \delta'^*\text{)}\end{aligned}$$

For each s in S, i in I and $o \in O$. We have:

$$\begin{aligned}&\delta(\langle \mathit{last}(s), \mathit{row}(s)\rangle, i)(\langle \mathit{last}(s \cdot i \cdot o), \mathit{row}(s \cdot i \cdot o)\rangle)\\ &\quad = T(s \cdot i)(o) && \text{(hypothesis construction)}\\ &\quad \text{and}\\ &\delta'(\phi(\langle \mathit{last}(s), \mathit{row}(s)\rangle), i)(\phi(\langle \mathit{last}(s \cdot i \cdot o), \mathit{row}(s \cdot i \cdot o)\rangle))\\ &\quad = \delta'(\delta'^*(q'_0, s), i)(\delta'^*(q'_0, s \cdot i \cdot o))\\ &\quad = M'(s \cdot i)(o) && \text{(Definition 7.1)}\\ &\quad = T(s \cdot i)(o) && (\mathcal{M}' \text{ is consistent with } T)\end{aligned}$$

Transition probabilities are preserved by the mapping ϕ.

Finally, we show that labelling is preserved. For all s in S:

$$\begin{aligned}L'(\phi(\langle \mathit{last}(s), \mathit{row}(s)\rangle)) &= L'(\delta'^*(q'_0, s))\\ &= \mathit{last}(s) && \text{(definition of } \delta'^*\text{)}\\ &\text{and}\\ L(\langle \mathit{last}(s), \mathit{row}(s)\rangle) &= \mathit{last}(s) && \text{(definition of } L\text{)}\\ \Rightarrow L'(\phi(\langle \mathit{last}(s), \mathit{row}(s)\rangle)) &= L(\langle \mathit{last}(s), \mathit{row}(s)\rangle)\end{aligned}$$

Labelling is preserved by the mapping ϕ.

Hence, ϕ is an isomorphism between $\mathcal{H}$ and $\mathcal{M}'$. □

This concludes the proof of Theorem 7.2. Lemma 7.6 shows consistency between hypotheses and the queried information stored in $\langle S, E, T\rangle$. From Lemma 7.7 and Lemma 7.8, we can infer that any MDP consistent with $\langle S, E, T\rangle$, but inequivalent to hypothesis $\mathrm{hyp}(S, E, T)$, must have more states than $\mathrm{hyp}(S, E, T)$.

Theorem 7.3 (Termination and Correctness).
The algorithm $L^*_{\mathrm{MDP}^e}$ terminates and returns an MDP $\mathcal{H}$ isomorphic to $\mathcal{M}$, thus it is minimal and it also satisfies $\mathcal{M} \equiv_{\mathrm{od}} \mathcal{H}$.

Proof.
Termination. Let $\langle S, E, T\rangle$ be a closed and consistent observation table and let $c \in \mathcal{TS}$ be a counterexample to equivalence between $\mathcal{M}$ and hypothesis $\mathrm{hyp}(S, E, T)$ with semantics H. Since c is a counterexample, $M(c) \neq H(c)$. Now let $\langle S', E', T'\rangle$ be an observation table extended by adding all prefix traces of c to S and (re-)establishing closedness and consistency. For $\mathrm{hyp}(S', E', T') = \mathcal{H}'$ with semantics H', we have $T'(c) = M(c)$ due to output distribution queries. Since $\mathcal{H}'$ is consistent with T', we have $T'(c) = H'(c) = M(c)$. Hence, $H'(c) \neq H(c)$, which shows that $\mathcal{H}'$ is not equivalent to $\mathcal{H}$, with c being a counterexample to equivalence. We do not remove elements from S, E, or T, thus $\mathcal{H}'$ is also consistent with T. Therefore, $\mathcal{H}'$ must have at least one state more than $\mathcal{H}$ according to Theorem 7.2. It follows that each round of learning, which finds a counterexample, adds at least one state. Since Algorithm 7.2 derives minimal hypotheses and $\mathcal{M}$ can be modelled with finitely many states, there can only be finitely many rounds that find counterexamples. Hence, we terminate after a finite number of rounds, because Algorithm 7.2 returns the final hypothesis as soon as an equivalence query $\mathbf{eq}$ returns *yes*, which happens once no counterexample can be found.

Correctness. The algorithm terminates when an equivalence query $\mathbf{eq}(\mathcal{H})$ does not find a counterexample between the final hypothesis $\mathcal{H}$ and $\mathcal{M}$. Since there is no counterexample at this point, we have $\mathcal{H} \equiv_{\mathrm{od}} \mathcal{M}$. Theorem 7.2 states that $\mathcal{H}$ is minimal and $\mathcal{M} = can(M)$ is consistent with T, because T is filled through output queries. Consequently, it follows from Lemma 7.8 that $\mathcal{H}$ is isomorphic to $\mathcal{M}$, the canonical MDP modelling the SUL. □

7.4 Learning MDPs by Sampling

In this section, we introduce L^*_{MDP}, a sampling-based learning algorithm for MDPs derived from $L^*_{\mathrm{MDP}^e}$. In contrast to $L^*_{\mathrm{MDP}^e}$, which requires exact information, L^*_{MDP} places weaker assumptions on the teacher. It does not require exact *output distribution* queries and *equivalence* queries, but approximates these queries via sampling by directed testing of the SUL. Distribution comparisons are consequently approximated through statistical tests. While we use similar data structures as in Section 7.3, we alter the learning algorithm structure. Since large amounts of data are required to produce accurate models, the sampling-based L^*_{MDP} allows to derive an approximate model at any time, unlike most other L^*-based algorithms. This section is split into three parts: first, we present a sampling-based interface between teacher and learner, as well as the interface between teacher and SUL. The second and third part describe the adapted learner and the implementation of the teacher, respectively.

7.4.1 Queries

The sampling-based teacher maintains a multiset of traces $\mathcal{S}$ for the estimation of output distributions. Whenever new traces are sampled in the course of learning, they are added to $\mathcal{S}$. In contrast to the exact setting, the teacher offers four queries, namely an equivalence query and three queries relating to output distributions and samples $\mathcal{S}$:

- *frequency query* ($\mathbf{fq}$): given a test sequence $s \in \mathcal{TS}$, $\mathbf{fq}(s) : O \to \mathbb{N}_0$ are output frequencies observed after s, where $\mathbf{fq}(s)(o) = \mathcal{S}(s \cdot o)$ for $o \in O$.
- *complete query* ($\mathbf{cq}$): given a test sequence $s \in \mathcal{TS}$, $\mathbf{cq}(s)$ returns $\mathbf{true}$ if sufficient information is available to estimate an output distribution from $\mathbf{fq}(s)$; returns $\mathbf{false}$ otherwise.

- *refine query* ($\mathbf{rfq}$): instructs the teacher to refine its knowledge of the SUL by testing it directed towards rarely observed samples. Traces sampled by $\mathbf{rfq}$ are added to $\mathcal{S}$, increasing the accuracy of subsequent probability estimations.
- *equivalence query* ($\mathbf{eq}$): given a hypothesis $\mathcal{H}$, $\mathbf{eq}$ tests for output-distribution equivalence between the SUL and $\mathcal{H}$; returns a counterexample from $\mathcal{TS}$ showing non-equivalence or returns *none* if no counterexample was found.

The sampling-based teacher thus needs to implement two different testing strategies, one for increasing accuracy of probability estimations along observed traces (*refine query*) and one for finding discrepancies between a hypothesis and the SUL (*equivalence query*). The *frequency query* and the *complete query* are used for hypothesis construction by the learner.

To test the SUL, we require the ability to (1) reset it and to (2) perform an input action and observe the produced output. For the remainder of this section, let $\mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$ be the canonical MDP underlying the SUL with semantics M. Based on $q \in Q$, the current execution state of $\mathcal{M}$, we define two operations available to the teacher:

- **reset:** resets $\mathcal{M}$ to the initial state by setting $q = q_0$ and returns $L(q_0)$.
- **step:** takes an input $i \in I$ and selects a new state q' according to $\delta(q,i)(q')$. The **step** operation then updates the execution state to q' and returns $L(q')$.

Note that we consider $\mathcal{M}$ to be a black box, assuming its structure and transition probabilities to be unknown. We are only able to perform inputs and observe output labels. For instance, we observe the initial SUL output $L(q_0)$ after performing a **reset**. The same two operations are used for sampling in Section 6.2 as well.

7.4.2 Learner Implementation

Sampling-Based Observation Tables

L^*_{MDP} also uses observation tables, therefore we use the same terminology as in Section 7.3. Sampling-based observation tables carry similar information as their exact counterparts, but instead of output distributions in $Dist(O)$, they store non-negative integers denoting observed output frequencies. More concretely, observation tables in the sampling-based setting store functions in $(O \to \mathbb{N}_0)$, from which we estimate probability distributions.

Definition 7.6 (Sampling-based Observation Table).
An observation table is a tuple $\langle S, E, \widehat{T}\rangle$, consisting of a prefix-closed set of traces $S \subseteq \mathcal{TR}$, a suffix-closed set of continuation sequences $E \subseteq \mathcal{CS}$, and a mapping $\widehat{T} : (S \cup Lt(S)) \cdot E \to (O \to \mathbb{N}_0)$, where $Lt(S) = \{s \cdot i \cdot o \,|\, s \in S, i \in I, o \in O : \mathbf{fq}(s \cdot i)(o) > 0\}$.

An observation table can be represented by a two-dimensional array, containing rows labelled with elements of S and $Lt(S)$ and columns labelled by E. Each table cell corresponds to a sequence $c = s \cdot e$, where $s \in S \cup Lt(S)$ is the row label of the cell and $e \in E$ is the column label. It stores queried output frequency counts $\widehat{T}(c) = \mathbf{fq}(c)$. To represent the content of rows, we define the function row on $S \cup Lt(S)$ by $row(s)(e) = \widehat{T}(s \cdot e)$. The traces in $Lt(S)$ are input-output-extensions of S which have been observed so far. We refer to traces in $S/Lt(S)$ as short/long traces. Analogously, we refer to rows labelled by corresponding traces as short and long rows.

As in Section 7.3, we identify states with traces reaching these states. These traces are stored in the prefix-closed set S. We distinguish states by their future behaviour in response to sequences in E. We initially set $S = \{L(q_0)\}$, where $L(q_0)$ is the initial output of the SUL, and $E = I$. Long traces, as extensions of access sequences in S, serve to define transitions of hypotheses.

Hypothesis Construction

As in Section 7.3, observation tables need to be closed and consistent for a hypothesis to be constructed. Unlike before, we do not have exact information to check equivalence of rows. We need to statistically test if rows are different. Therefore, Definition 7.7 gives a condition to determine whether two sequences lead to statistically different observations, meaning that the corresponding output frequency samples come from different distributions. The condition is based on Hoeffding bounds [142] which are also used by Carrasco and Oncina [75]. In Definition 7.8, we further apply this condition in a check for approximate equivalence between cells and extend this check to rows. Using similar terminology as Carrasco and Oncina [75], we refer to such checks as compatibility checks and we say that two cells/rows are compatible if we determine that they are not statistically different. These notions of compatibility serve as the basis for slightly adapted definitions of closedness and consistency.

The confidence levels of the statistical tests in Condition 2.b. of Definition 7.7 depend on a parameter α [75], which we assume to be globally accessible to simplify presentation. For convergence proofs, we let α depend on the number of sampled traces in $\mathcal{S}$, but we observed slightly better performance for small constant values, such as 0.05; see also Section 7.6. In general, learning performance can be assumed to be robust with respect to the exact setting of α. Carrasco and Oncina pointed out that: "The algorithm behaved robustly with respect to the choice of parameter α, due to its logarithmic dependence on the parameter." [75] Similar observations have been made by Mao et al. [199], who applied Hoeffding bounds as well.

Definition 7.7 (Different).
Two test sequences s and s' in $\mathcal{TS}$ produce statistically different output distributions with respect to $f : \mathcal{TS} \to (O \to \mathbb{N}_0)$, denoted $\mathit{diff}_f(s, s')$, iff

1. $\mathbf{cq}(s) \wedge \mathbf{cq}(s') \wedge n_1 > 0 \wedge n_2 > 0$[2]*, where $n_1 = \sum_{o \in O} f(s)(o)$, $n_2 = \sum_{o \in O} f(s')(o)$, and one of the following conditions holds*

2.a. $\exists o \in O : \neg(f(s)(o) > 0 \Leftrightarrow f(s')(o) > 0)$*, or*

2.b. $\exists o \in O : \left|\frac{f(s)(o)}{n_1} - \frac{f(s')(o)}{n_2}\right| > \left(\sqrt{\frac{1}{n_1}} + \sqrt{\frac{1}{n_2}}\right) \cdot \sqrt{\frac{1}{2} \ln \frac{2}{\alpha}}$*, where α specifies the confidence level $(1-\alpha)^2$ for testing each o separately based on a Hoeffding bound [75, 142].*

Definition 7.8 (Compatible).
Two cells labelled by $c = s \cdot e$ and $c' = s' \cdot e'$ are compatible, denoted $\mathbf{compatible}(c, c')$, iff $\neg \mathit{diff}_{\widehat{T}}(c, c')$. Two rows labelled by s and s' are compatible, denoted $\mathbf{compatible}_E(s, s')$ iff $\mathit{last}(s) = \mathit{last}(s')$ and the cells corresponding to all $e \in E$ are compatible, i.e. $\forall e \in E : \mathbf{compatible}(s \cdot e, s' \cdot e)$.

Compatibility Classes

In Section 7.3, we formed equivalence classes of traces with respect to $\mathbf{eqRow}_E$ to create one hypothesis state per equivalence class. Now we partition rows labelled by S based on compatibility. Compatibility given by Definition 7.8, however, is not an equivalence relation, as it is not transitive in general. As a result, we cannot simply create equivalence classes.

We apply the heuristic implemented by Algorithm 7.3 to partition S. This heuristic chooses a representative r for each block in the partition based on the amount of information available for r. The available information is computed by the rank function in Algorithm 7.3, which basically checks how many input-output extensions of r have been sampled. As in equivalence classes, every trace in a block is compatible to its block representative.

[2] A similar condition is also used in the compatibility checks by Carrasco and Oncina [75], which assumes no difference if there are no observations for either s or s'.

Algorithm 7.3 Creation of compatibility classes

```
1: for all $s \in S$ do
2:     $\mathrm{rank}(s) \leftarrow \sum_{i \in I} \sum_{o \in O} \widehat{T}(s \cdot i)(o)$
3: end for
4: $\mathit{unpartitioned} \leftarrow S$
5: $R \leftarrow \emptyset$
6: while $\mathit{unpartitioned} \neq \emptyset$ do
7:     $r \leftarrow m$ where $m \in \mathit{unpartitioned}$ with largest $\mathrm{rank}(m)$
8:     $R \leftarrow R \cup \{r\}$
9:     $\mathit{cg}(r) \leftarrow \{s \in \mathit{unpartitioned} \mid \mathbf{compatible}_E(s, r)\}$
10:    for all $s \in \mathit{cg}(r)$ do
11:        $\mathit{rep}(s) \leftarrow r$
12:    end for
13:    $\mathit{unpartitioned} \leftarrow \mathit{unpartitioned} \setminus \mathit{cg}(r)$
14: end while
```

Algorithm 7.3 first assigns a rank to each trace in S. Then, it partitions S by iteratively selecting the trace r with the largest rank and computing a *compatibility class* $\mathit{cg}(r)$ for r. The trace r is the (canonical) representative for s in $\mathit{cg}(r)$, which we denote by $\mathit{rep}(s)$ (Line 11). Each representative r is stored in the set of representative traces R. In contrast to equivalence classes, elements in a compatibility class need not be pairwise compatible and an s may be compatible to multiple representatives, where the unique representative $\mathit{rep}(s)$ of s has the largest rank. However, in the limit $\mathbf{compatible}_E$ based on Hoeffding bounds converges to an equivalence relation [75] and therefore compatibility classes are equivalence classes in the limit; see Section 7.5.

Definition 7.9 (Sampling Closedness).
An observation table $\langle S, E, \widehat{T}\rangle$ is closed if for all $l \in Lt(S)$ there is a representative $s \in R$ with $\mathbf{compatible}_E(l, s)$.

Definition 7.10 (Sampling Consistency).
An observation table $\langle S, E, \widehat{T}\rangle$ is consistent if for all compatible pairs of short traces s, s' in S and all input-output pairs $i{\cdot}o \in I{\cdot}O$, we have that (1) at least one of the extensions has not been observed yet, i.e. $\widehat{T}(s{\cdot}i)(o) = 0$ or $\widehat{T}(s'{\cdot}i)(o) = 0$, or (2) both extensions are compatible, i.e. $\mathbf{compatible}_E(s{\cdot}i{\cdot}o, s'{\cdot}i{\cdot}o)$.

The adapted notions of closedness and consistency of observation tables are based on compatibility and compatibility classes. Note that the first condition of consistency may be satisfied because of incomplete information.

Hypothesis Construction

Given a closed and consistent observation table $\langle S, E, \widehat{T}\rangle$, we derive a hypothesis $\mathcal{H} = \mathrm{hyp}(S, E, \widehat{T})$ through the steps below. Note that extensions $s \cdot i \cdot o$ of s in $R \subseteq S$ define transitions. Some extensions may have few observations, such that $\sum_{o \in O} \widehat{T}(s \cdot i)(o)$ is low and $\mathbf{cq}(s \cdot i) = \mathbf{false}$. In case of such uncertainties, we add transitions to a special sink state labelled by "chaos", an output not in the original alphabet[3]. A hypothesis is a tuple $\mathcal{H} = \langle Q_\mathrm{h}, I, O \cup \{\mathrm{chaos}\}, q_{0\mathrm{h}}, \delta_\mathrm{h}, L_\mathrm{h}\rangle$ where:

[3]This is inspired by the introduction of chaos states in **ioco**-based learning [283]. We also added such a state in Section 6.2, if the trace sample for learning did not include information on certain state-input pairs.

- representatives for long traces $l \in Lt(S)$ are given by (see Algorithm 7.3): $rep(l) = r$ where $r \in \{r' \in R \mid \mathbf{compatible}_E(l, r')\}$ with largest $\mathrm{rank}(r)$
- $Q_\mathrm{h} = \{\langle last(s), row(s)\rangle \mid s \in R\} \cup \{q_\mathrm{chaos}\}$,
  - for $q = \langle o, row(s)\rangle \in Q_\mathrm{h} \setminus \{q_\mathrm{chaos}\}$: $L_\mathrm{h}(q) = o$
  - for q_chaos: $L_\mathrm{h}(q_\mathrm{chaos}) = \mathrm{chaos}$ and for all $i \in I$: $\delta_\mathrm{h}(q_\mathrm{chaos}, i)(q_\mathrm{chaos}) = 1$
- $q_{0\mathrm{h}} = \langle L(q_0), row(L(q_0))\rangle$ (note that $L(q_0) \in S$ due to initialisation)
- for $q = \langle last(s), row(s)\rangle \in Q_\mathrm{h} \setminus \{q_\mathrm{chaos}\}$ and $i \in I$ (note that $I \subseteq E$):
  1. If $\neg\mathbf{cq}(s \cdot i)$: $\delta(q, i)(q_\mathrm{chaos}) = 1$, i.e. move to chaos
  2. Otherwise estimate a distribution $\mu = \delta_\mathrm{h}(q, i)$ over the successor states: for $o \in O$ with $\widehat{T}(s \cdot i)(o) > 0$: $\mu(\langle o, row(rep(s \cdot i \cdot o))\rangle) = \frac{\widehat{T}(s \cdot i)(o)}{\sum_{o' \in O} \widehat{T}(s \cdot i)(o')}$

Updating the Observation Table

Closedness and Consistency. Analogously to Section 7.3, we make observation tables closed by adding new short rows and we establish consistency by adding new columns. We refer to the function implementing that also as MAKECLOSEDANDCONSISTENT. While $L^*_{\mathrm{MDP}^e}$ implemented by Algorithm 7.2 needs to fill the observation table after executing MAKECLOSEDANDCONSISTENT, this is not required in the sampling-based setting due to the adapted notions of closedness and consistency.

Trimming the Observation Table. Observation table size greatly affects learning performance, therefore it is common to avoid adding redundant information [151, 240], which we also discussed in Section 2.3. Due to inexact information, this is hard to apply in a stochastic setting. We instead remove rows via a function TRIM once we are certain that removing them does not change the hypothesis. Given an observation table $\langle S, E, \widehat{T}\rangle$, we remove s and all s' such that $s \ll s'$ from S if:

1. there is exactly one $r \in R$ such that $\mathbf{compatible}_E(s, r)$
2. $s \notin R$ and $\forall r \in R : \neg(s \ll r)$
3. and $\forall s' \in S, i \in I$, with $s \ll s'$: $\mathit{diff}_{\mathbf{fq}}(s' \cdot i, r \cdot i) = \mathbf{false}$, where $r \in R$ such that $\delta^*_\mathrm{h}(r) = \delta^*_\mathrm{h}(s')$, $\langle last(r), row(r)\rangle = \delta^*_\mathrm{h}(r)$, and δ_h is the transition relation of $\mathrm{hyp}(S, E, \widehat{T})$.

The first condition is motivated by the observation that if s is compatible to exactly one r, then all extensions of s can be assumed to reach the same states as the extensions of r. In this case, s presumably corresponds to the same SUL state as r, therefore we do not need to store s in the observation table. The other conditions make sure that we do not remove required rows, because of a spurious compatibility check in the first condition. The second condition ensures that we do not remove representatives and the third condition is related to the implementation of equivalence queries. It basically checks for compatibility between the hypothesis $\mathrm{hyp}(S, E, \widehat{T})$ and the frequency information available via *frequency* queries. This is done by checking if an extension s' of s reveals a difference between the output frequencies observed after s' (queried via $\mathbf{fq}$) and the output frequencies observed after the representative r reaching the same hypothesis state as s'. Put differently, we check if s' is a counterexample to equivalence between hypothesis and SUL. Note that removed rows do not affect hypothesis construction.

Algorithm 7.4 The main algorithm implementing the sampling-based L^*_{MDP}

```
Input: sampling-based teacher capable of answering fq, rfq, eq and cq
Output: final learned model $\mathrm{hyp}(S, E, \widehat{T})$
 1: $S \leftarrow \{L(q_0)\}, E \leftarrow I, \widehat{T} \leftarrow \{\}$          ▷ initialise observation table
 2: perform rfq($\langle S, E, \widehat{T}\rangle$)          ▷ sample traces for initial observation table
 3: for all $s \in S \cup Lt(S), e \in E$ do
 4:     $\widehat{T}(s \cdot e) \leftarrow$ fq($s \cdot e$)          ▷ update observation table with frequency information
 5: end for
 6: $round \leftarrow 0$
 7: repeat
 8:     $round \leftarrow round + 1$
 9:     while $\langle S, E, \widehat{T}\rangle$ not closed or not consistent do
10:         $\langle S, E, \widehat{T}\rangle \leftarrow$ MAKECLOSEDANDCONSISTENT($\langle S, E, \widehat{T}\rangle$)
11:     end while
12:     $\mathcal{H} \leftarrow \mathrm{hyp}(S, E, \widehat{T})$          ▷ create hypothesis
13:     $\langle S, E, \widehat{T}\rangle \leftarrow$ TRIM($\langle S, E, \widehat{T}\rangle, \mathcal{H}$)          ▷ remove rows that are not needed
14:     $cex \leftarrow$ eq($\mathcal{H}$)
15:     if $cex \neq none$ then          ▷ we found a counterexample
16:         for all $(t \cdot i) \in prefixes(cex)$ with $i \in I$ do
17:             $S \leftarrow S \cup \{t\}$          ▷ add all prefixes of the counterexample
18:         end for
19:     end if
20:     perform rfq($\langle S, E, \widehat{T}\rangle$)          ▷ sample traces to refine knowledge about SUL
21:     for all $s \in S \cup Lt(S), e \in E$ do
22:         $\widehat{T}(s \cdot e) \leftarrow$ fq($s \cdot e$)          ▷ update observation table with frequency information
23:     end for
24: until STOP($\langle S, E, \widehat{T}\rangle, \mathcal{H}, round$)
25: return $\mathrm{hyp}(S, E, \widehat{T})$          ▷ output final hypothesis
```

Learning Algorithm

Algorithm 7.4 implements L^*_{MDP}. It first initialises an observation table $\langle S, E, \widehat{T}\rangle$ with the initial SUL output as first row and with the inputs I as columns (Line 1). Lines 2 to 5 perform a refine query and then update $\langle S, E, \widehat{T}\rangle$ with output frequency information, which corresponds to output distribution queries in $L^*_{\text{MDP}^e}$. Here, the teacher resamples the only known trace $L(q_0)$. Resampling that trace consists of observing $L(q_0)$, performing some input, and observing another output.

After that, we perform Line 7 to Line 24 until a stopping criterion is reached. We establish closedness and consistency of $\langle S, E, \widehat{T}\rangle$ in Line 10 to build a hypothesis $\mathcal{H}$ in Line 12. After that, we remove redundant rows of the observation table via TRIM in Line 13. Then, we perform an equivalence query, testing for equivalence between SUL and $\mathcal{H}$. If we find a counterexample, we add all its prefix traces as rows to the observation table as in $L^*_{\text{MDP}^e}$. Finally, we sample new system traces via **rfq** to gain more accurate information about the SUL (Lines 20 to 23). Once we stop, we output the final hypothesis.

Stopping. $L^*_{\text{MDP}^e}$ and deterministic automata learning usually stop learning once equivalence between the learned hypothesis and the SUL is achieved. In implementations of equivalence queries via testing, equivalence is usually assumed to hold when no counterexample can be found. Here, we employ a different stopping criterion, because equivalence can hardly be achieved via sampling. Furthermore, we may wish to carry on resampling via **rfq** although we did not find a counterexample. Resampling may improve the accuracy of a hypothesis such that subsequent equivalence queries reveal counterexamples.

Our stopping criterion takes statistical uncertainty[4] in compatibility checks into account. As previously noted, rows may be compatible to multiple other rows. In particular, a row labelled by s may be compatible to multiple representatives. In such a case, we are not certain which state is reached by the trace s. We address this issue by stopping based on the ratio r_{unamb} of unambiguous traces to all traces, which we compute by:

$$r_{\mathrm{unamb}} = \frac{|\{s \in S \cup Lt(S) : \mathrm{unambiguous}(s)\}|}{|S \cup Lt(S)|} \text{ where}$$
$$\mathrm{unambiguous}(s) \Leftrightarrow |\{r \in R : \mathbf{compatible}_E(s,r)\}| = 1$$

More concretely, we stop if:

1.a. at least $r_{\min}$ rounds have been executed and

1.b. the chaos state q_{chaos} is unreachable and

1.c. and $r_{\mathrm{unamb}} \geq t_{\mathrm{unamb}}$, where t_{unamb} is a user-defined threshold,

or

2.a. alternatively we stop after a maximum number of rounds $r_{\max}$.

7.4.3 Teacher Implementation

In the following, we describe the implementation of each of the four queries provided by the teacher. Recall that we interact with the SUL with semantics M via the two operations $\mathbf{reset}$ and $\mathbf{step}$; see Section 6.2 and Section 7.4.1. The canonical MDP $can(M) = \mathcal{M} = \langle Q, I, O, q_0, \delta, L\rangle$ has the same observable behaviour as the SUL.

Frequency Query

The teacher keeps track of a multiset of sampled system traces $\mathcal{S}$. Whenever a new a trace is added, all its prefixes are added as well, as they have been observed as well. Therefore, we have for $t \in \mathcal{TR}, t' \in$ *prefixes*$(t) : \mathcal{S}(t) \leq \mathcal{S}(t')$. The frequency query $\mathbf{fq}(s) : O \to \mathbb{N}_0$ for $s \in \mathcal{TS}$ returns output frequencies observed after s:

$$\forall o \in O : \mathbf{fq}(s)(o) = \mathcal{S}(s \cdot o)$$

Complete Query

Trace frequencies retrieved via $\mathbf{fq}$ are generally used to compute empirical output distributions μ following a sequence s in $\mathcal{TS}$, i.e. the learner computes $\mu(o) = \frac{\mathbf{fq}(s)(o)}{\sum_{o' \in O} \mathbf{fq}(s)(o')}$ to approximate $M(s)(o)$. The *complete* query $\mathbf{cq}$ takes a sequence s as input and signals whether s should be used to approximate $M(s)$, for instance, to perform statistical tests[5]. We base $\mathbf{cq}$ on a threshold $n_c > 0$ by defining:

$$\mathbf{cq}(s) = \begin{cases} \mathbf{true} & \text{if } \sum_{o \in O} \mathcal{S}(s \cdot o) \geq n_c \\ \mathbf{true} & \text{if } \exists s', o, i : s' \cdot o \cdot i \ll s \wedge \mathbf{cq}(s') \wedge \mathcal{S}(s \cdot o) = 0 \\ \mathbf{false} & \text{otherwise} \end{cases}$$

Note that for a complete s, all prefixes of s are also complete. Additionally if $\mathbf{cq}(s)$, then we assume that we have seen all extensions of s. Therefore, we set for each o with $\mathcal{S}(s \cdot o) = 0$ all extensions of $s \cdot o$ to be complete (second clause). The threshold n_c is user-specifiable in our implementation.

[4]This form of uncertainty does not refer to uncertain behaviour in general, but to uncertainty related to decisions based on statistical tests.

[5]This query serves a similar purpose as the *completeness queries* used by Volpato and Tretmans [283].

Algorithm 7.5 *Refine* query

```
Input: observation table $\langle S, E, \widehat{T} \rangle$, number of requested new traces $n_{\mathrm{resample}}$
Output: updated multiset of traces $\mathcal{S}$
 1: $rare \leftarrow \{s \mid s \in (S \cup Lt(S)) \cdot E : \neg \mathbf{cq}(s)\}$          ▷ identify incomplete sequences
 2: $prefixTree \leftarrow$ BUILDPREFIXTREE$(rare)$
 3: for $i \leftarrow 1$ to $n_{\mathrm{resample}}$ do          ▷ collect $n_{\mathrm{resample}}$ new samples
 4:     $newTrace \leftarrow$ SAMPLESUL$(prefixTree)$
 5:     $\mathcal{S} \leftarrow \mathcal{S} \uplus \{newTrace\}$
 6: end for
 7: function SAMPLESUL$(prefixTree)$
 8:     $node \leftarrow root(prefixTree)$
 9:     $trace \leftarrow \mathbf{reset}$          ▷ initialise SUL and observe initial output
10:     loop
11:         $input \leftarrow rSel(\{i \in I \mid \exists o \in O, n : node \xrightarrow{i,o} n\})$          ▷ random input
12:         $output \leftarrow \mathbf{step}(i)$          ▷ execute SUL and observe output
13:         $trace \leftarrow trace \cdot i \cdot o$
14:         if $trace \notin prefixTree$ or $trace$ labels leaf then          ▷ did we leave the tree?
15:             return $trace$
16:         end if
17:         $node' \leftarrow n$ with $node \xrightarrow{i,o} n$          ▷ move in tree
18:         $node \leftarrow node'$
19:     end loop
20: end function
```

Refine Query

Refine queries serve the purpose of refining our knowledge about output distributions along previously observed traces. Therefore, we select rarely observed traces and resample them to perform this query. We implemented this through the procedure outlined in Algorithm 7.5.

First, we build a prefix tree from rarely observed traces (Line 1 and Line 2), where edges are labelled by input-output pairs and nodes are labelled by traces reaching the nodes. This tree is then used for directed online-testing of the SUL via SAMPLESUL (Lines 7 to 20) with the goal of reaching a leaf of the tree. In this way, we create n_{resample} new samples and add them to the multiset of samples $\mathcal{S}$.

Equivalence Query

As explained in Section 2.4, equivalence queries are often implemented via (conformance) testing in active automata, for instance, using the W-method [83] method for deterministic models. Such testing techniques generally execute some test suite to find counterexamples to conformance between a model and the SUL. In our setup, a counterexample is a test sequence inducing a different output distribution in the hypothesis $\mathcal{H}$ than in the SUL. Since we cannot directly observe those distributions, we perform equivalence queries by applying two strategies to find counterexamples. First, we search for counterexamples with respect to the structure of $\mathcal{H}$ via testing. Secondly, we check for statistical conformance between all traces $\mathcal{S}$ sampled so far and $\mathcal{H}$, which allows us to detect incorrect output distributions.

Note that all traces to the state q_{chaos} are guaranteed to be counterexamples, as its label chaos is not part of the original output alphabet O. For this reason, we do not search for other counterexamples if q_{chaos} is reachable in $\mathcal{H}$. In slight abuse of terminology, we implement this by returning *none* from $\mathbf{eq}(\mathcal{H})$. L^*_{MDP} in Algorithm 7.4 will then issue further $\mathbf{rfq}$ queries, lowering uncertainty about state transitions, which in turn causes q_{chaos} to be unreachable eventually.

Algorithm 7.6 State-coverage-based testing for counterexample detection

```
Input: $\mathcal{H} = \langle Q_\mathrm{h}, I, O_\mathrm{h}, q_{0\mathrm{h}}, \delta_\mathrm{h}, L_\mathrm{h} \rangle$, schedulers $qSched$
Output: counterexample test sequence $s \in \mathcal{TS}$ or $none$
 1: $q_\mathrm{curr} \leftarrow q_{0\mathrm{h}}$                                        ▷ current state
 2: $trace \leftarrow \mathbf{reset}$
 3: $q_\mathrm{target} \leftarrow rSel(reachable(Q_\mathrm{h}, q_\mathrm{curr}))$    ▷ choose a target state
 4: loop
 5:     if $coinFlip(p_\mathrm{rand})$ then
 6:         $in \leftarrow rSel(I)$                                      ▷ random next input
 7:     else
 8:         $in \leftarrow qSched(q_\mathrm{target})$          ▷ next input leads towards target
 9:     end if
10:     $out \leftarrow \mathbf{step}(in)$                                      ▷ perform input
11:     $q_\mathrm{curr} \leftarrow \Delta_\mathrm{h}(q_\mathrm{curr}, in \cdot out)$                   ▷ move in hypothesis
12:     if $q_\mathrm{curr} = \perp$ then                     ▷ output not possible in hypothesis
13:         return $trace \cdot in$                              ▷ return counterexample
14:     end if
15:     $trace \leftarrow trace \cdot in \cdot out$
16:     if $coinFlip(p_\mathrm{stop})$ then                  ▷ stop with probability $p_\mathrm{stop}$
17:         return $none$
18:     end if
19:     if $q_\mathrm{curr} = q_\mathrm{target}$ or $q_\mathrm{target} \notin reachable(Q_\mathrm{h}, q_\mathrm{curr})$ then
20:         $q_\mathrm{target} \leftarrow rSel(reachable(Q_\mathrm{h}, q_\mathrm{curr}))$        ▷ choose new target
21:     end if
22: end loop
```

Testing of Structure. Our goal in testing is to sample a trace of the SUL that is not observable on the hypothesis. For this purpose, we adapted the randomised *transition coverage* testing strategy, which we presented in Chapter 4, from Mealy machines to MDPs. We adapted this strategy rather than the *mutation* strategy, because it is unclear how to adapt the *mutation* strategy and our evaluation in Section 4.4 showed that *transition coverage* was effective in most cases [17]. It reliably found counterexamples with a low number of tests. Since MDPs produce outputs in states, whereas Mealy machines produce outputs on transitions, we target *state coverage* in MDP-based testing instead of transition coverage.

To test stochastic SULs, we generate test cases that interleave random walks in hypotheses with paths leading to randomly chosen states. By performing many of these tests, we aim at covering hypotheses adequately, while exploring new parts of the SUL's state space through random testing. In contrast to Section 4.2, but as in Chapter 6, we perform online testing. This is necessary, because of the stochastic behaviour of the SUL. The online-testing procedure is outlined in Algorithm 7.6.

The algorithm takes a hypothesis and *qSched* as input where *qSched* is a mapping from states to schedulers. Given $q \in Q$, *qSched*(q) is a scheduler maximising the probability of reaching q, therefore it selects inputs optimally with respect to the reachability of q. As noted in Section 5.3.1, there exist memoryless and deterministic schedulers for optimal reachability [114]. Such schedulers take only the last state in the current execution path into account and they select inputs non-probabilistically. Hence, a scheduler *qSched*(q) is a function $s : Q \to I$. We compute these schedulers similarly as in Section 6.2, except that here we consider unbounded reachability of a hypothesis state, rather than bounded reachability of an output. Unfortunately, value iteration is less efficient and accurate in this scenario.

In Algorithm 7.6, we start by randomly choosing a target state q_target from the states reachable from the initial state (Line 3). These states are given by $reachable(Q, q_\mathrm{curr})$. Then, we execute the SUL, either with random inputs (Line 6) or with inputs leading to the target (Line 8), which are computed using schedulers. If we observe an output which is not possible in the hypothesis, we return a counterexample

(Line 13). Alternatively, we may stop with probability p_{stop} (Line 17). If we reach the target or it becomes unreachable, we simply choose a new target state (Line 20).

For each equivalence query, we repeat Algorithm 7.6 up to n_{test} times and report the first counterexample that we find, if any. In case we find a counterexample c, we resample it up to n_{retest} times or until $\mathbf{cq}(c)$, to get more accurate information about it. If we do not find a counterexample, we check for conformance between the current hypothesis and the sampled traces $\mathcal{S}$, as described below.

Checking Conformance to $\mathcal{S}$. For each sequence $t \cdot i \in \mathcal{TS}$ with $i \in I$ such that $\mathbf{cq}(t \cdot i)$, we check for consistency between the information stored in $\mathcal{S}$ and the current hypothesis $\mathcal{H}$ by evaluating two conditions:

1. Is t observable on $\mathcal{H}$? If it is not, then we determine the longest observable prefix t' of t such that $t' \cdot i' \cdot v = t$, where i' is a single input, and return $t' \cdot i'$ as counterexample from $\mathbf{eq}(\mathcal{H})$. The sequence $t' \cdot i'$ is a counterexample, because at least one of its extensions has been observed ($\mathbf{cq}(t \cdot i) \implies \mathcal{S}(t) > 0$), but none of its extensions is observable on the hypothesis.

2. If t is observable, we determine $q = \langle o, row(r) \rangle$ reached by t in $\mathcal{H}$, where $r \in R$, and return $t \cdot i$ as counterexample if $\mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$ is true. This statistical check approximates the comparison $M(t \cdot i) \neq M(r \cdot i)$, to check if $t \not\equiv_M r$. Therefore, it implicitly checks $M(t \cdot i) \neq H(t \cdot i)$, as $t \equiv_H r$.

If neither of the two equivalence checking strategies, i.e. testing of structure and checking conformance to $\mathcal{S}$, finds a counterexample, we return $none$ from the corresponding equivalence query.

7.5 Convergence of L^*_{MDP}

In the following, we will show that the sampling-based L^*_{MDP} learns a correct MDP. Based on the notion of language identification in grammatical inference [95], we describe our goal as producing an MDP isomorphic to the canonical MDP modelling the SUL with probability one in the limit.

To show identification in the limit, we introduce slight simplifications. First, we disable trimming of the observation table (see Section 7.4.2), thus we do not remove rows. Secondly, we set $p_{\text{rand}} = 1$ for equivalence testing and we do not stop testing at the first detected difference between SUL and hypothesis, but we stop solely based on a $p_{\text{stop}} < 1$. As a result, all input choices are distributed uniformly at random (i.e. equivalence testing does not use schedulers for achieving state coverage) and the length of each test is geometrically distributed with p_{stop}. This is motivated by the common assumption that sampling distributions do not change during learning [95]. Thirdly, we change the function rank in Algorithm 7.3 to assign ranks based on a lexicographic ordering of traces rather than based on observed frequencies, such that the trace consisting only of the initial SUL output has the highest rank. We actually implemented both types of rank functions and found that the frequency-based function led to better accuracy, but would require more complex proofs. We let the number of samples for learning approach infinity, therefore we do not use a stopping criterion. Finally, we concretely instantiate the complete query $\mathbf{cq}$ by setting $n_c = 1$, since n_c is only relevant for applications in practice.

7.5.1 Proof Structure

We show convergence in two major steps: (1) first we show that the hypothesis structure derived from a sampling-based observation table converges to the hypothesis structure derived from the corresponding observation table with exact information. (2) Then, we show that if counterexamples exist, we will eventually find them. Through that, we eventually arrive at a hypothesis with the same structure as the canonical MDP $can(M)$, where M is the SUL semantics. Given a hypothesis with correct structure, it follows by the law of large numbers that the estimated transition probabilities converge to the true probabilities, thus the hypotheses converge to an MDP isomorphic to $can(M)$.

A key point of the proofs concerns the convergence of the statistical test applied by diff_f, which is based on Hoeffding bounds [142]. With regard to that, we apply similar arguments as Carrasco and Oncina [75, p.11-13 & Appendix]. Given convergence of diff_f, we also rely on the convergence of the exact learning algorithm $L^*_{\mathrm{MDP}^e}$ discussed in Section 7.3.4. Another important point is that the shortest traces in each equivalence class of $S/{\equiv_M}$ do not form loops in $can(M)$. Hence, there are finitely many such traces. Furthermore, for a given $can(M)$ and some hypothesis MDP, the shortest counterexample has bounded length, therefore it suffices to check finitely many test sequences to check for overall equivalence.

Definitions & Notation

We show convergence in the limit of the number of sampled system traces n. We take n into account through a data-dependent α_n for the Hoeffding bounds used by diff_f defined in Definition 7.7. More concretely, let $\alpha_n = \frac{1}{n^r}$ for $r > 2$ as used by Mao et al. [199], which implies $\sum_n \alpha_n n < \infty$. For the remainder of this section, let $\langle S_n, E_n, \widehat{T}_n \rangle$ be the closed and consistent observation table containing the first n samples stored by the teacher in the multiset $\mathcal{S}_n$. Furthermore, let $\mathcal{H}_n$ be the hypothesis $\mathrm{hyp}(S_n, E_n, \widehat{T}_n)$, let the semantics of the SUL be M and let $\mathcal{M}$ be the canonical MDP $can(M)$. We say that two MDPs have the same structure, if their underlying graphs are isomorphic, thus exact transition probabilities may be different. Now we can state the main theorem on convergence.

Theorem 7.4 (Convergence).
Given a data-dependent $\alpha_n = \frac{1}{n^r}$ for $r > 2$, which implies $\sum_n \alpha_n n < \infty$, then with probability one, the hypothesis $\mathcal{H}_n$ is isomorphic to $\mathcal{M}$, except for finitely many n.

Hence, in the limit, we learn an MDP that is minimal with respect to the number of states and output-distribution equivalent to the SUL.

Access Sequences

The exact learning algorithm $L^*_{\mathrm{MDP}^e}$ presented in Section 7.3 iteratively updates an observation table. Upon termination it arrives at an observation table $\langle S, E, T \rangle$ and a hypothesis $\mathcal{H} = \mathrm{hyp}(S, E, T) = \langle Q_{\mathrm{h}}, I, O, q_{0\mathrm{h}}, \delta_{\mathrm{h}}, L_{\mathrm{h}} \rangle$. Let $S_{\mathrm{acc}} \subseteq S$ be the set of shortest access sequences leading to states in Q given by $S_{\mathrm{acc}} = \{s \mid s \in S, \nexists s' \in S : s' \ll s \wedge s' \neq s \wedge \delta^*_{\mathrm{h}}(q_{0\mathrm{h}}, s) = \delta^*_{\mathrm{h}}(q_{0\mathrm{h}}, s')\}$ (the shortest traces in each equivalence class of $S/\mathbf{eqRow}_E = S/{\equiv_M}$). By this definition, S_{acc} forms a directed spanning tree in the structure of $\mathcal{H}$. There are finitely many different spanning trees for a given hypothesis, therefore there are finitely many different S_{acc}. Hypothesis models learned by $L^*_{\mathrm{MDP}^e}$ are isomorphic to $\mathcal{M}$, thus there are finitely many possible final hypotheses. Let $\overline{S}$ be the finite union of all access sequence sets S_{acc} forming spanning trees in all valid final hypotheses. Let $\overline{L} = \{s \cdot i \cdot o \mid s \in \overline{S}, i \in I, o \in O, M(s \cdot i)(o) > 0\}$ be one-step extensions of $\overline{S}$ with non-zero probability.

Note that for the construction of correct hypotheses in $L^*_{\mathrm{MDP}^e}$, it is sufficient for $\mathbf{eqRow}_E$ to approximate M-equivalence (see Definition 7.2) for traces in $\overline{L}$. Actually, if $\mathbf{eqRow}_E$ is equivalent to M-equivalence for traces in $\overline{L}$, then both are equivalent on all possible traces in $\mathcal{TR}$. Consequently, the approximation of $\mathbf{eqRow}_E$ via $\mathbf{compatible}_E$ needs to hold only for traces in $\overline{L}$.

Proof. We now show that $\forall t_1, t_2 \in \mathcal{TR} : \mathbf{eqRow}_E(t_1, t_2) \Leftrightarrow t_1 \equiv_M t_2$ follows from $\forall t'_1, t'_2 \in \overline{L} : \mathbf{eqRow}_E(t'_1, t'_2) \Leftrightarrow t'_1 \equiv_M t'_2$.

Direction "$\Leftarrow$": By the definition of $\equiv_M$ and $\mathbf{eqRow}_E$, we have $\mathbf{eqRow}_E(t_1, t_2) \Leftarrow t_1 \equiv_M t_2$ for all $t_1, t_2 \in \mathcal{TR}$ and all $E \subseteq \mathcal{CS}$. Intuitively, two traces in the same M-equivalence class cannot be distinguished by a finite set of continuation sequences.

Direction "⇒": Hence, it remains to show that $\forall t_1, t_2 \in \mathcal{TR} : \mathbf{eqRow}_E(t_1,t_2) \Rightarrow t_1 \equiv_M t_2$ if $\forall t'_1, t'_2 \in \overline{L} : \mathbf{eqRow}_E(t'_1,t'_2) \Leftrightarrow t'_1 \equiv_M t'_2$. We shall prove this by contradiction, thus we assume that there exist t_1 and t_2 in $\mathcal{TR}$ such that $\mathbf{eqRow}_E(t_1,t_2)$, but $t_1 \not\equiv_M t_2$, thus violating the implication.

Let t'_1 and t'_2 be traces in $\overline{L}$ such that $t'_1 \equiv_M t_1$ and $t'_2 \equiv_M t_2$. These traces exist, because $\overline{L}$ contains all input-output extensions of traces corresponding to simple paths in the structure of the canonical MDP $\mathcal{M}$. Put differently, all states can be reached by traces in $\overline{L}$. By our assumption and transitivity of $\equiv_M$, we have $t'_1 \not\equiv_M t'_2$. Since $t'_1, t'_2 \in \overline{L}$ and $\mathbf{eqRow}_E(t'_1,t'_2) \Leftrightarrow t'_1 \equiv_M t'_2$ for such traces, it follows that $\neg\mathbf{eqRow}_E(t'_1,t'_2)$.

Since $t'_1 \equiv_M t_1$ and $t'_2 \equiv_M t_2$, we have $\mathbf{eqRow}_E(t'_1,t_1)$ and $\mathbf{eqRow}_E(t'_2,t_2)$ (see Direction "⇐"). Due to the transitivity of $\mathbf{eqRow}_E$ and the assumption that $\mathbf{eqRow}_E(t_1,t_2)$ holds, we can deduce $\mathbf{eqRow}_E(t'_1,t'_2)$. This is a contradiction to the paragraph above. Hence, our assumption must be wrong. There does not exist a pair of traces $t_1, t_2 \in \mathcal{TR}$ such that $\mathbf{eqRow}_E(t_1,t_2)$, but $t_1 \not\equiv_M t_2$. □

7.5.2 Hoeffding-Bound-Based Difference Check

Before discussing hypothesis construction, we briefly discuss the Hoeffding-bound-based test applied by diff_f. Recall that for two test sequences s and s', we test for each $o \in O$ if the probability p of observing o after s is different than the probability p' of observing o after s'. This is implemented through:

$$\exists o \in O : \left| \frac{f(s)(o)}{n_1} - \frac{f(s')(o)}{n_2} \right| > \left(\sqrt{\frac{1}{n_1}} + \sqrt{\frac{1}{n_2}} \right) \cdot \sqrt{\frac{1}{2} \ln \frac{2}{\alpha}} = \epsilon_\alpha(n_1, n_2)$$

As pointed out by Carrasco and Oncina [75, p.11-13 & Appendix], this test works with a confidence level above $(1-\alpha)^2$ and for large enough n_1 and n_2, it tests for difference and equivalence of p and p'. More concretely, for convergence, n_1 and n_2 must be such that $2\epsilon_\alpha(n_1,n_2)$ is smaller than the smallest absolute difference between any two different p and p'. As our data-dependent α_n decreases only polynomially, $\epsilon_\alpha(n_1,n_2)$ tends to zero for increasing n_1 and n_2. Hence, the test implemented by diff_f converges to an exact comparison between p and p'.

In the remainder of the paper, we ignore Condition 2.a for diff_f, which checks if the sampled distributions have the same support. By applying a data-dependent α_n, as defined above, Condition 2.b converges to an exact comparison between output distributions, thus 2.a is a consequence of 2.b in the limit. Therefore, we consider only the Hoeffding-bound-based tests of Condition 2.b.

7.5.3 Hypothesis Construction

Theorem 7.5 (Compatibility Convergence).
Given α_n such that $\sum_n \alpha_n n < \infty$, then with probability one: $\mathbf{compatible}_E(s,s') \Leftrightarrow \mathbf{eqRow}_E(s,s')$ for all traces s, s' in $\overline{L}$, except for finitely many n.

Proof. Let A_n be the event that $\mathbf{compatible}_E(s,s') \not\Leftrightarrow \mathbf{eqRow}_E(s,s')$ and $p(A_n)$ be the probability of this event. In the following, we derive a bound for $p(A_n)$ based on the confidence level of applied tests in Definition 7.7 which is above $(1-\alpha_n)^2$ [75]. An observation table stores $|S \cup Lt(S)| \cdot |E|$ cells, which gives us an upper bound on the number of tests performed for computing $\mathbf{compatible}_E(s,s')$ for two traces s and s'. However, note that cells do not store unique information; multiple cells may correspond to the same test sequence in $\mathcal{TS}$, therefore it is simpler to reason about the number of different tests in calls to $\mathit{diff}_{\widehat{T}}(c,c') = \mathit{diff}_{\mathbf{fq}}(c,c')$ with respect to $\mathcal{S}_n$. A single call to $\mathit{diff}_{\mathbf{fq}}$ involves either 0 or $|O|$ tests. We apply tests only if we have observed both c and c' at least once, therefore we perform at most $2 \cdot |O| \cdot n$ different tests for all pairs of observed test sequences. The event A_n may occur if any test produces an incorrect result, i.e. it yields a Boolean result different from the comparison between the true output distributions induced by c and c'. This leads to $p(A_n) \leq 2 \cdot |O| \cdot n \cdot (1-(1-\alpha_n)^2)$, which implies $p(A_n) \leq 4 \cdot |O| \cdot n \cdot \alpha_n$. By choosing α_n such that $\sum_n \alpha_n n < \infty$, we have $\sum_n p(A_n) < \infty$ and we can apply the Borel-Cantelli lemma like Carrasco and Oncina [75], which states A_n happens only finitely often. Hence, there is an N_{comp} such that for $n > N_{\text{comp}}$, we have $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$ with respect to $\mathcal{S}_n$. □

Lemma 7.9.
Under the assumed uniformly randomised equivalence testing strategy, it holds for every $s \cdot i \cdot o \in \overline{L}$: $\mathcal{S}_n(s \cdot i \cdot o) > 0$ after finitely many n.

Proof. Informally, we will eventually sample all traces $l \in \overline{L}$. The probability p_L of sampling $l = o_0 \cdot i_1 \cdot o_1 \cdots o_n \cdot i \cdot o$ during a test, where $l[\leq k]$ is the prefix test sequence of l of length k, is given by (note that we may sample l as a prefix of another sequence):

$$p_L = \frac{1}{|I|^{n+1}} M(l[\leq 1])(o_1) \cdots M(l[\leq n])(o_n) \cdot M(l[\leq n+1])(o)(1 - p_{\text{stop}})^n$$

Since every $l \in \overline{L}$ is observable, we have $M(l[\leq 1])(o_1) \cdots M(l[\leq n])(o_n) \cdot M(t[\leq n+1])(o) > 0$, thus $p_L > 0$. Hence, there is a finite N_L such that for all $s \cdot i \cdot o \in \overline{L} : \mathcal{S}_n(s \cdot i \cdot o) > 0$ for $n > N_L$. □

Lemma 7.10.
If $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$, then the set of representatives R computed by Algorithm 7.3 for the closed and consistent observation table $\langle S_n, E_n, \widehat{T}_n \rangle$ is prefix-closed.

Proof. Recall that we assume the function rank to impose a lexicographic ordering on traces, thus all ranks are unique. This simplifies showing prefix-closedness of R, which we do by contradiction. Assume that R is not prefix-closed. In that case, there is a trace r of length n in R with a prefix r_p of length $n - 1$ that is not in R. Since $r_p \notin R$ and because the representative $rep(r_p)$ has the largest rank in its class $cg(r_p)$, we have $r_p \neq rep(r_p)$ and $\text{rank}(r_p) < \text{rank}(rep(r_p))$.

As S_n is prefix-closed and $R \subseteq S_n$, we have $r_p \in S_n$. Let $i \in I$ and $o \in O$ be such that $r_p \cdot i \cdot o = r$. Algorithm 7.3 enforces $\mathbf{compatible}_E(r_p, rep(r_p))$ and due to consistency, we have that $\mathbf{compatible}_E(r_p \cdot i \cdot o, rep(r_p) \cdot i \cdot o) = \mathbf{compatible}_E(r, rep(r_p) \cdot i \cdot o)$. Since r is a representative in R, $rep(r_p) \cdot i \cdot o \in cg(r)$. Representatives r have the largest rank in their compatibility class $cg(r)$ and $r \neq rep(r_p) \cdot i \cdot o$, thus $\text{rank}(r) > \text{rank}(rep(r_p) \cdot i \cdot o)$.

In combination we have that

$$\text{rank}(r_p) < \text{rank}(rep(r_p)) \text{ and also}$$
$$\text{rank}(r) = \text{rank}(r_p \cdot i \cdot o) > \text{rank}(rep(r_p) \cdot i \cdot o).$$

This is a contradiction given the lexicographic ordering on traces imposed by rank. Consequently, R must be prefix-closed under the premises of Lemma 7.10. □

Lemma 7.11.
Let $\langle S_n, E_n, T_n \rangle$ be the exact *observation table corresponding to the sampling-based observation table $\langle S_n, E_n, \widehat{T}_n \rangle$, i.e. $T_n(s) = \mathbf{odq}(s)$ for $s \in (S_n \cup Lt(S_n)) \cdot E$. Then, $T_n(r \cdot i)(o) > 0 \Leftrightarrow \widehat{T}_n(r \cdot i)(o) > 0$ for $r \in R, i \in I, o \in O$ after finitely many n.*

Proof. First, we will show for prefix-closed R (Lemma 7.10) that $R \subseteq \overline{S}$, if $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$. $\overline{S}$ contains all traces corresponding to simple paths of $can(M)$, therefore we show by contradiction that no $r \in R$ forms a cycle in $can(M)$.

Assume that $r \in R$ forms a cycle in $can(M)$, i.e. it visits states multiple times. We can split r into three parts $r = r_p \cdot r_c \cdot r_s$, where $r_p \in \mathcal{TR}$ such that r_p and $r_p \cdot r_c$ reach the same state, r_c is non-empty, and $r_s \in (I \times O)^*$ is the longest suffix such that starting from $\delta^*(r_p)$, r_s visits every state of $can(M)$ at most once. As R is prefix-closed, R includes r_p and $r_p \cdot r_c$ as well. The traces r_p and $r_p \cdot r_c$

reach the same state in $can(M)$, thus we have $r_p \equiv_M r_p \cdot r_c$ which implies $\mathbf{eqRow}_E(r_p, r_p \cdot r_c)$ and $\mathbf{compatible}_E(r_p, r_p \cdot r_c)$.

By Algorithm 7.3 all $r \in R$ are pairwise not compatible with respect to $\mathbf{compatible}_E$, thus Algorithm 7.3 ensures $\neg\mathbf{compatible}_E(r_p, r_p \cdot r_c)$ for $r_p \in R$ and $r_p \cdot r_c \in R$. This leads to a contradiction, therefore our assumption is false and we can deduce that no r visits a state of $can(M)$ more than once. As a consequence, it holds that $R \subseteq \overline{S}$.

Hence, every observable $r_l = r \cdot i \cdot o$ for $r \in R, i \in I$ and $o \in O$ is in $\overline{L}$, as $\overline{L}$ includes all observable extensions of $\overline{S}$. By Lemma 7.9, we will sample r_l eventually, i.e. $\widehat{T}_n(r \cdot i)(o) > 0$ and therefore $T_n(r \cdot i)(o) > 0 \Leftrightarrow \widehat{T}_n(r \cdot i)(o) > 0$ after finitely many n.

□

Lemma 7.12.
The chaos state q_{chaos} is not reachable in $\mathcal{H}_n$, except for finitely many n.

Proof. We add a transition from state $q = \langle last(r), row(r)\rangle$ with input i to q_{chaos} if $\mathbf{cq}(r \cdot i) = \mathbf{false}$. As we consider $n_c = 1$, $\mathbf{cq}(r \cdot i) = \mathbf{true}$ if there is an o such that $\widehat{T}_n(r \cdot i)(o) > 0$. Lemma 7.11 states that $\widehat{T}_n(r \cdot i)(o) > 0$ for any observable $r \cdot i \cdot o$ after finitely many n. Thus, Lemma 7.11 implies $\mathbf{cq}(r \cdot i) = \mathbf{true}$ for all $r \in R$ and $i \in I$, therefore the chaos is unreachable in $\mathcal{H}_n$, except for finitely many n. □

Corollary 7.1.
Let $\langle S_n, E_n, T_n\rangle$ be the exact observation table corresponding to the sampling-based observation table $\langle S_n, E_n, \widehat{T}_n\rangle$, i.e. $T_n(s) = \mathbf{odq}(s)$ for $s \in (S_n \cup Lt(S_n)) \cdot E$. Then there exists a finite N_{struct} such that the exact hypothesis $\text{hyp}(S_n, E_n, T_n)$ has the same structure as $\mathcal{H}_n$ for $n > N_{\text{struct}}$.

By combining Theorem 7.5, Lemma 7.11 and Lemma 7.12, it follows that, after finitely many n, hypotheses created in the sampling-based setting have the same structure as in the exact setting.

7.5.4 Equivalence Queries

Theorem 7.6 (Convergence of Equivalence Queries).
Given α_n such that $\sum_n \alpha_n n < \infty$, an observation table $\langle S_n, E_n, \widehat{T}_n\rangle$ and a hypothesis $\mathcal{H}_n$, then with probability one, $\mathcal{H}_n$ has the same structure as $\mathcal{M}$ or we find a counterexample to equivalence, except for finitely many n.

In the following, we introduce lemmas to prove Theorem 7.6, but first we recap relevant details of equivalence queries. According to Corollary 7.1, there is an N_{struct} such that $\mathcal{H}_n$ has the same structure as in the exact setting and $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$ for $n > N_{\text{struct}}$. Therefore, we assume $n > N_{\text{struct}}$ for the following discussion of counterexample search through the implemented equivalence queries $\mathbf{eq}$. Let H_n be the semantics of $\mathcal{H}_n$. Recall that we apply two strategies for checking equivalence:

1. Random testing with a uniformly randomised scheduler ($p_{\text{rand}} = 1$): this form of testing can find traces $s \cdot o$, with $s \in \mathcal{TS}$ and $o \in O$, such that $H(s)(o) = 0$ and $M(s)(o) > 0$. While this form of search is coarse, we store all sampled traces in $\mathcal{S}_n$ that is used by our second counterexample search strategy for performing a fine-grained analysis.

2. Checking conformance with $\mathcal{S}_n$: for all observed test sequences, we statistically check for differences between output distributions in $\mathcal{H}_n$ and distributions estimated from $\mathcal{S}_n$ through applying *diff*$_{\mathbf{fq}}$. Applying that strategy finds counterexample sequences $s \in \mathcal{TS}$ such that $M(s) \neq \bot$ (as s must have been observed) and approximately $M(s) \neq H(s)$.

Case 1. In the case that the hypothesis $\mathcal{H}_n$ and $\mathcal{M}$ have the same structure and $n > N_{\text{struct}}$, such that $\mathbf{eqRow}_E(s, s') \Leftrightarrow \mathbf{compatible}_E(s, s')$, we may still find counterexamples that are spurious due to inaccuracies. Therefore, we will show that adding a prefix-closed set of traces to the set of short traces S_n does not change the hypothesis structure, as this is performed by Algorithm 7.4 in response to counterexamples returned by $\mathbf{eq}$.

Lemma 7.13.
If $\mathcal{H}_n$ has the same structure as $\mathcal{M}$ and $n > N_{\text{struct}}$, then adding a prefix-closed set of observable traces S_t to S_n will neither introduce closedness-violations nor inconsistencies, hence $\langle S_n \cup S_t, E_n, \widehat{T_n}\rangle$ is closed and consistent. Consequently, the hypothesis structure does not change, thus $\mathcal{H}_n$ has the same structure as $\text{hyp}(S_n \cup S_t, E_n, \widehat{T_n})$.

Proof. Let t be a trace in S_t and $q_t = \delta^*_{\text{h}}(t)$ be the hypothesis state reached by t, which exists because $\mathcal{H}_n$ has the same structure as $\mathcal{M}$. Let $t_s \in S_n$ be a short trace also reaching q_t. Since $\mathcal{M}$ and $\mathcal{H}_n$ have the same structure, t and t_s also reach the same state of $\mathcal{M}$, therefore $t \equiv_M t_s$ (by reaching the same state both traces lead to the same future behaviour), which implies $\mathbf{eqRow}_E(t, t_s)$. With $n > N_{\text{struct}}$, we have $\mathbf{compatible}_E(t, t_s)$. By the same reasoning, we have $\mathbf{compatible}_E(t \cdot i \cdot o, t_s \cdot i \cdot o)$ for any $i \in I, o \in O$ with $M(t \cdot i)(o) > 0$; which is the condition for consistency of observation tables, therefore adding t to S_n leaves the observation tables consistent.

Furthermore because $\langle S_n, E_n, \widehat{T_n}\rangle$ is closed, there exists a $t'_s \in S_n$, with $\mathbf{compatible}_E(t_s \cdot i \cdot o, t'_s)$. Since $\mathbf{compatible}_E(t \cdot i \cdot o, t_s \cdot i \cdot o)$ and because $\mathbf{compatible}_E$ is transitive for $n > N_{\text{struct}}$, we have $\mathbf{compatible}_E(t \cdot i \cdot o, t'_s)$. Hence, adding t as to S_n does not violate closedness, because for each observable extensions of t, there exists a compatible short trace t'_s.

□

Case 2. If $\mathcal{H}_n$ does not have the same structure as $\mathcal{M}$ and $n > N_{\text{struct}}$, then $\mathcal{H}_n$ has fewer states than $\mathcal{M}$. This follows from Lemma 7.7 given that $\mathcal{H}$ is consistent with $\widehat{T_n}$ and $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$. Since $\mathcal{M}$ is minimal with respect to the number of states, $\mathcal{H}_n$ and $\mathcal{M}$ are not equivalent, thus a counterexample to observation equivalence exists and we are guaranteed to find any such counterexample after finitely many samples.

Lemma 7.14.
If $\mathbf{compatible}_E(s, s') \Leftrightarrow \mathbf{eqRow}_E(s, s')$ for traces s and s' in S_n, then the hypothesis $\mathcal{H}_n$ derived from $\langle S_n, E_n, \widehat{T_n}\rangle$ has the minimal number of states among all MDPs consistent with $\widehat{T_n}$ with respect to diff$_{\widehat{T_n}}$.

Proof. Recall that for a given observation table $\langle S, E, T\rangle$, the exact learning algorithm $L^*_{\text{MDP}^e}$ derives the smallest hypothesis consistent with T. By Corollary 7.1, $\mathcal{H}_n$ has the same structure as the smallest MDP consistent with T. As *diff*$_{\widehat{T_n}}$ does not produce spurious results for $n > N_{\text{struct}}$ (Theorem 7.5), $\mathcal{H}_n$ has the same structure as the smallest MDP consistent with $\widehat{T_n}$ with respect to *diff*$_{\widehat{T_n}}$, therefore the number of states of $\mathcal{H}_n$ is minimal.

□

Lemma 7.15.
Let n_q be the number of states of $\mathcal{M}$, $C = \bigcup_{i=0}^{n_q^2+1} (O \times I)^i$ and $C^{\text{obs}} = \{c \mid c \in C : M(c) \neq \bot\}$. For any other MDP $\mathcal{M}'$ with at most n_q states and semantics M', we have $\forall c \in C^{\text{obs}} : M(c) = M'(c)$ iff $\mathcal{M} \equiv_{\text{od}} \mathcal{M}'$.

Hence, there is a finite set C^{obs} of sequences with lengths bounded by $n_q^2 + 1$ such that by testing all sequences in C^{obs}, we can check equivalence with certainty.

Proof. Let $\mathcal{M}$ and $\mathcal{M}'$ with states Q and Q' be defined as above, i.e. $|Q| = n_q$ and $|Q'| \leq n_q$, and let $\mathrm{reachQSeq}(t) \in (Q \times Q')^*$ be the sequence of state pairs visited along a trace t by $\mathcal{M}$ and $\mathcal{M}'$, respectively. $\mathcal{M} \equiv_{\text{od}} \mathcal{M}'$ iff for all $t \in \mathcal{TR}$ and $i \in I$, we have $M(t \cdot i) = M'(t \cdot i)$. If the length of $t \cdot i$ is at most $n_q^2 + 1$, then $t \cdot i \in C$. Otherwise, $\mathrm{reachQSeq}(t)$ contains duplicated state pairs, because $|Q \times Q'| \leq n_q^2$. For t longer than n_q^2, we can remove loops on $Q \times Q'$ from t to determine a trace t' of length at most n_q^2 such that $\mathrm{reachQSeq}(t)[|t|] = \mathrm{reachQSeq}(t')[|t'|]$, i.e. such that t and t' reach the same state pair. Since t reaches the same state as t' in $\mathcal{M}$ and in $\mathcal{M}'$, we have $M(t \cdot i) = M(t' \cdot i)$ and $M'(t \cdot i) = M'(t' \cdot i)$, thus $M(t \cdot i) = M'(t \cdot i) \Leftrightarrow M(t' \cdot i) = M'(t' \cdot i)$. Consequently for all $t \cdot i \in \mathcal{TR} \cdot I$: either $t \cdot i \in C$, or there is a $t' \cdot i \in C$ leading to an equivalent check between $\mathcal{M}$ and $\mathcal{M}'$.

We further restrict C to C^{obs} by considering only observable test sequences in C. This restriction is justified by Lemma 7.2. In summary:

$$\begin{aligned}
\mathcal{M} \equiv_{\text{od}} \mathcal{M}' &\Leftrightarrow \forall c \in \mathcal{TS} : M(c) = M'(c) \\
&\Leftrightarrow \forall c \in C : M(c) = M'(c) \\
&\Leftrightarrow \forall c \in C^{\text{obs}} : M(c) = M'(c)
\end{aligned}$$

□

Lemma 7.16.
Under the randomised testing strategy with $p_{\text{rand}} = 1$ and $p_{\text{stop}} < 1$, all c in C^{obs} have non-zero probability to be observed.

Proof. Due to $p_{\text{rand}} = 1$ and $p_{\text{stop}} < 1$ we apply uniformly randomised inputs during testing and each test has a length that is distributed dependent on p_{stop}. Let $c = o_0 \cdot i_1 \cdot o_1 \cdots o_{n-1} \cdot i_n$ be a sequence in C^{obs} with $c[\leq k]$ being its prefix of length k, then the probability p_c of observing c is (note that we may observe c as a prefix of another sequence):

$$p_c = \frac{1}{|I|^n} M(c[\leq 1])(o_1) \cdot M(c[\leq 2])(o_2) \cdots M(c[\leq n-1])(o_{n-1}) \cdot (1 - p_{\text{stop}})^{n-1}$$

By definition of C^{obs}, we have $M(c[\leq j])(o_j) > 0$ for all indexes j and c in C^{obs}, therefore $p_c > 0$. □

In every round of L^*_{MDP}, we check for conformance between $\mathcal{S}_n$ and the hypothesis $\mathcal{H}_n$ and return a counterexample, if we detect a difference via $\mathit{diff}_{\mathbf{fq}}$. Since we apply $\mathit{diff}_{\mathbf{fq}}$, we follow a similar reasoning as for the convergence of hypothesis construction. Here, we approximate $M(c) \neq H(c)$ for $c \in \mathcal{TS}$ by $\mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$, where $c = t \cdot i$ for a trace t, an input i, and $r \in R$ given by the hypothesis state $\langle \mathit{last}(r), \mathit{row}(r) \rangle$ reached by t.

Lemma 7.17.
Given α_n such that $\sum_n \alpha_n n < \infty$, then with probability one $M(c) \neq H(c) \Leftrightarrow \mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$ for $c = t \cdot i \in C^{\text{obs}}$ and r as defined above, except for finitely many n.

Proof. We use the identity $H(t \cdot i) = H(r \cdot i)$ for traces t and r and inputs i, which holds because t and r reach the same state in the hypothesis $\mathcal{H}$. Applying that, we test for $M(t \cdot i) \neq H(t \cdot i)$ by testing $M(t \cdot i) \neq H(r \cdot i)$ via $\mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$. We perform $|O|$ tests for each unique observed sequence c, therefore we apply at most $n \cdot |O|$ tests. Let B_n be the event that any of these tests is wrong, that is, $M(t \cdot i) \neq H(r \cdot i) \not\Leftrightarrow \mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$ for at least one observed $c = t \cdot i$. Due to the confidence level greater than $(1 - \alpha_n)^2$ of the tests, the probability $p(B_n)$ of B_n is bounded by $p(B_n) \leq n \cdot |O| \cdot (1 - (1 - \alpha_n)^2) \leq 2 \cdot n \cdot |O| \cdot \alpha_n$. By choosing α_n such that $\sum_n \alpha_n n < \infty$, we can apply the Borel-Cantelli lemma as in the proof of Theorem 7.5. Hence, B_n happens only finitely often, thus there is an N_1 such that for all $n > N_1$ we have $M(t \cdot i) \neq H(r \cdot i) \Leftrightarrow \mathit{diff}_{\mathbf{fq}}(t \cdot i, r \cdot i)$ for all observed $c = t \cdot i \in C^{\text{obs}}$. Furthermore, the probability of observing any c of the finite set C^{obs} during testing is greater than zero (Lemma 7.16), thus there is a finite N_2 such that $\mathcal{S}_n$ contains all $c \in C^{\text{obs}}$ for $n > N_2$. Consequently, there is an N_{cex}, such that Lemma 7.17 holds for all $n > N_{\text{cex}}$. □

Lemma 7.14 states that hypotheses $\mathcal{H}_n$ are minimal after finitely many n, thus all potential counterexamples are in C^{obs} (Lemma 7.15). From Lemma 7.17, it follows that we will identify a counterexample in C^{obs} if one exists. Combining that with Lemma 7.13 concludes the proof of Theorem 7.6.

7.5.5 Putting Everything Together

We have established that after finitely many n, the sampling-based hypothesis $\mathcal{H}_n$ has the same structure as in the exact setting (Corollary 7.1). Therefore, certain properties of the exact learning algorithm $L^*_{\text{MDP}^e}$ hold for the sampling-based L^*_{MDP} as well. The derived hypotheses are therefore minimal, i.e. they have at most as many states as $\mathcal{M}$. As with $L^*_{\text{MDP}^e}$, adding a non-spurious counterexample to the trace set S_n introduces at least one state in the derived hypotheses. Furthermore, we have shown that equivalence queries return non-spurious counterexamples, except for finitely many n (Theorem 7.6). Consequently, after finite n we arrive at a hypothesis $\mathcal{H}_n$ with the same structure as $\mathcal{M}$. We derive transition probabilities by computing empirical means, thus by the law of large numbers these estimated probabilities converge to the true probabilities. Hence, we learn a hypothesis $\mathcal{H}_n$ isomorphic to the canonical MDP $\mathcal{M}$ in the limit as stated by Theorem 7.4.

More efficient parameters. So far, we discussed a particular parametrisation of L^*_{MDP}. Among others, we used uniformly random input choices for equivalence testing with $p_{\text{rand}} = 1$, and instantiated **cq** to accept samples as complete after only $n_c = 1$ observation. This simplified the proof, but is inefficient in practical experiments. As demonstrated in Section 4.5, completely random testing is inefficient for large systems and a small n_c may lead to spurious states in intermediate hypotheses. However, the arguments based on $n_c = 1$, such as Lemma 7.11 and Lemma 7.12, are easily extended to small constant values of n_c. Since the samples are collected independently, any observation that occurs at least once after a finite number of steps also occurs at least n_c times after a finite number of steps.

7.6 Experiments

In active automata learning, our goal is generally to learn a model equivalent to the true model of the SUL. This changes in the stochastic setting, where we want to learn a model close to the true model, because equivalence can hardly be achieved. In this section, we evaluate the sampling-based L^*_{MDP} and compare it to the passive IoAlergia [199], by learning several models with both techniques. In each case, we treat the known true MDP model $\mathcal{M}$ as a black box for learning and measure similarity to this model using two criteria:

1. *bisimilarity distance:* we compute the discounted bisimilarity distance between the true model $\mathcal{M}$ and the learned MDPs [44, 45]. Giovanni Bacci adapted the distance measure from MDPs with rewards to labelled MDPs, by defining a distance of 1 between states with different labels.
2. *probabilistic model-checking:* we compute and compare maximal probabilities of manually defined temporal properties with all models using Prism 4.4 [174]. The difference between probabilistic model-checking results for $\mathcal{M}$ and the learned MDPs serves as a similarity measure.

We performed experiments with three of the models that we used in the evaluation of probabilistic black-box reachability checking in Section 6.3, including the gridworld, the slot machine, and the shared coin consensus protocol. In addition to that, we performed experiments with another larger gridworld model. Experimental results, the examined models, and the implementation of L^*_{MDP} can be found in the evaluation material [261].

Table 7.2: Results for learning models of the first gridworld

	true model	L^*_{MDP}	IoAlergia
# outputs	-	3,101,959	3,103,607
# traces	-	391,530	387,746
time [s]	-	118.4	21.4
# states	35	35	21
$\delta_{0.9}$	-	0.1442	0.5241
$\mathbb{P}_{\max}(F^{\leq 11}(\text{goal}))$	0.9622	0.9651	0.2306
$\mathbb{P}_{\max}(\neg\text{G}\, U^{\leq 14}(\text{goal}))$	0.6499	0.6461	0.1577
$\mathbb{P}_{\max}(\neg\text{S}\, U^{\leq 16}(\text{goal}))$	0.6912	0.6768	0.1800

7.6.1 Measurement Setup

Like Mao et al. [199] did in one of two configurations, we configure IoAlergia with a data-dependent significance parameter for the compatibility check, by setting $\epsilon_N = \frac{10000}{N}$, where N is the total combined length of all traces used for learning. This parameter serves a purpose analogous to the α parameter for the Hoeffding bounds used by L^*_{MDP}. IoAlergia showed good performance under this configuration in the experiments discussed in this section. Other than Section 6.3, where we used a fixed ϵ, this section also considers unbounded properties and bisimilarity distances. In contrast to that, we observed that L^*_{MDP} generally shows better performance with non-data-dependent α, therefore we set $\alpha = 0.05$ for all experiments. We sample traces for IoAlergia with a length geometrically distributed with parameter p_l and inputs chosen uniformly at random, like Mao et al. [199]. The number of traces is chosen such that IoAlergia and L^*_{MDP} learn from approximately the same amount of data.

We implemented L^*_{MDP} and IoAlergia using Java 8. In addition to our Java implementations, we use Prism 4.4 [174] for probabilistic model-checking and an adaptation of the MDPDist library [43] to labelled MDPs for computing bisimilarity distances. We performed the experiments with a Lenovo Thinkpad T450 running Xubuntu Linux 18.04 with 16 GB RAM and an Intel Core i7-5600U CPU with 2.6 GHz.

7.6.2 Experiments with First Gridworld

The basis for our first experiments is the gridworld that we used in Section 6.3 to evaluate our learning-based testing technique. It is shown in Figure 6.6a. As before, a robot moves around in this world of tiles of different terrains. It may make errors in movement, for instance, by moving south west instead of south with an error probability depending on the target terrain. Our aim in this chapter is to learn an environment model, i.e. a map, rather than a strategy for reaching a goal location as in Section 6.3. The minimal true model of this gridworld has 35 different states.

For L^*_{MDP}, we set the sampling parameters to $n_{\text{resample}} = n_{\text{retest}} = 300$, $n_{\text{test}} = 50$, $p_{\text{stop}} = 0.25$ and $p_{\text{rand}} = 0.25$. As stopping parameters served $t_{\text{unamb}} = 0.99$, $r_{\min} = 500$ and $r_{\max} = 4000$. Finally, we set the parameter p_l for IoAlergia's geometric trace length distribution to 0.125.

Results. Table 7.2 shows the measurement results for learning the first gridworld. Active learning stopped after 1147 rounds, sampling 391,530 traces (Row 2) with a combined number of outputs of $3{,}101{,}959$ (Row 1). The bisimilarity distance discounted with $\lambda = 0.9$ to the true model is 0.144 for L^*_{MDP} and 0.524 for IoAlergia (Row 5); thus it can be assumed that model checking the L^*_{MDP} model produces more accurate results. This is indeed true for our three evaluation queries in the last three rows. These model-checking queries ask for the maximum probability (quantified over all schedulers) of reaching the goal within a varying number of steps. The first query does not restrict the terrain visited before the goal, but the second and third require to avoid G and S, respectively. The absolute difference

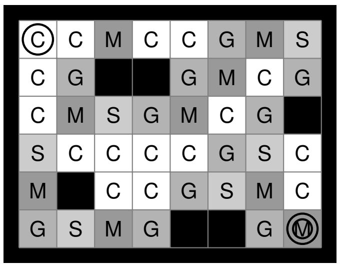


Figure 7.1: The second evaluation gridworld

Table 7.3: Results for learning models of the second gridworld

	true model	L^*_{MDP}	IOALERGIA
# outputs	-	3,663,415	3,665,746
# traces	-	515,950	457,927
time [s]	-	166.9	15.1
# states	72	72	31
$\delta_{0.9}$	-	0.1121	0.5763
$\mathbb{P}_{\max}(F^{\leq 14}(\mathrm{goal}))$	0.9348	0.9404	0.0208
$\mathbb{P}_{\max}(F^{\leq 12}(\mathrm{goal}))$	0.6712	0.6796	0.0172
$\mathbb{P}_{\max}(\neg\mathrm{M}\ U^{\leq 18}(\mathrm{goal}))$	0.9743	0.9750	0.0196
$\mathbb{P}_{\max}(\neg\mathrm{S}\ U^{\leq 20}(\mathrm{goal}))$	0.1424	0.1644	0.0240

to the true values is at most 0.015 for L^*_{MDP}, but the results for IOALERGIA differ greatly from the true values. One reason for this is that the IOALERGIA model with 21 states is significantly smaller than the minimal true model, while the L^*_{MDP} model has as many states as the true model.

IOALERGIA is faster than L^*_{MDP}, which applies time-consuming computations during equivalence queries. However, as noted in Section 6.3 and in Section 4.4, the runtime of learning-specific computations is often negligible in practical applications. This is due to the fact that the communication with the SUL during test-case execution usually dominates the overall runtime. We experienced that while learning models of MQTT brokers which we discuss in Section 3.4.5 [263].

Given the smaller bisimilarity distance and the lower difference to the true probabilities computed with PRISM, we conclude that the L^*_{MDP} model is more accurate.

7.6.3 Experiments with Second Gridworld

Figure 7.1 shows the second gridworld used in our evaluation. As before, the robot starts in the initial location in the top left corner and can only observe the different terrains. The goal location is in the bottom right corner in this example. The true minimal MDP representing this gridworld has 72 states. We configured learning as for the first gridworld, but collect more samples per round by setting $n_{\mathrm{retest}} = n_{\mathrm{resample}} = 1{,}000$. Table 7.3 shows the measurement results including the model-checking results and the bisimilarity distances.

L^*_{MDP} sampled $515{,}950$ traces with a combined number of outputs of $3{,}663{,}415$. Hence, the combined length of all traces is in a similar range as before, although L^*_{MDP} sampled more traces in a single round. This is the case, because learning stopped already after 500 rounds. We used similar model-checking queries as in the previous example that ask for the maximum probability of reaching the goal location with varying number of steps. The third and fourth query additionally specify to avoid the terrains M and the S, respectively. We can again see that the L^*_{MDP} model has the same size as the minimal true model, while the IOALERGIA model is much smaller. The bisimilarity difference between the true model and the L^*_{MDP} model is much smaller than for IOALERGIA as well. Also as in the first gridworld experiment,

Table 7.4: Results for learning models of the shared coin consensus-protocol

	true	L^*_{MDP}	IoAlergia
# outputs	-	537,665	537,885
# traces	-	98,064	67,208
time [s]	-	3,188.9	3.5
# states	272	163	94
$\delta_{0.9}$	-	0.1142	0.4482
$\mathbb{P}_{\max}(F(\text{finished} \land \text{p}_1\text{\_heads} \land \text{p}_2\text{\_tails}))$	0.1069	0	0
$\mathbb{P}_{\max}(F(\text{finished} \land \text{p}_1\text{\_tails} \land \text{p}_2\text{\_tails}))$	0.5556	0.6765	0.6594
$\mathbb{P}_{\max}(\text{counter} \neq 5\ U \text{ finished})$	0.3333	0.3899	0.5356
$\mathbb{P}_{\max}(\text{counter} \neq 4\ U \text{ finished})$	0.4286	0.5191	0.6682
$\mathbb{P}_{\max}(F^{<40}(\text{finished} \land \text{p}_1\text{\_heads} \land \text{p}_2\text{\_tails}))$	0.0017	0	0
$\mathbb{P}_{\max}(F^{<40}(\text{finished} \land \text{p}_1\text{\_tails} \land \text{p}_2\text{\_tails}))$	0.2668	0.3066	0.2694
$\mathbb{P}_{\max}(\text{counter} \neq 5\ U^{<40} \text{ finished})$	0.2444	0.2928	0.4460
$\mathbb{P}_{\max}(\text{counter} \neq 4\ U^{<40} \text{ finished})$	0.2634	0.3246	0.5050

the difference to the true model-checking results is smaller for L^*_{MDP}. However, compared to the previous example, the absolute difference between the L^*_{MDP} model and the true model with respect to model-checking has slightly increased.

7.6.4 Shared Coin Consensus-Protocol Experiments

The following experiments are based on the shared coin consensus protocol by Aspnes and Herlihy [42] that we also examined in Section 6.3. Recall that we adapted a model of the protocol distributed with the PRISM model checker [174]. We generally performed only minor adaptions such as adding action labels for inputs, but we also slightly changed the functionality by doing that. For the purpose of this evaluation these changes are immaterial, though.

As in Section 6.3, we consider the configuration with the smallest state space of size 272 with two processes and the constant K set to 2. We set the learning parameters to $n_{\text{resample}} = n_{\text{retest}} = n_{\text{test}} = 50$, $p_{\text{stop}} = 0.25$ and $p_{\text{rand}} = 0.25$. We controlled stopping with $t_{\text{unamb}} = 0.99$, $r_{\min} = 500$ and $r_{\max} = 4000$. Finally, we set $p_l = 0.125$ for IoAlergia.

Table 7.4 shows the measurement results computed in the evaluation of learned models of the shared coin consensus protocol. Compared to the previous examples, we need a significantly lower sample size of 98,064 traces containing 537,665 outputs, although the models are much larger. A reason for this is that there is a relatively large number of outputs in this example, such that states are easier to distinguish from each other. The bisimilarity distance is in a similar range as before for L^*_{MDP}, which is again significantly smaller than IoAlergia's bisimilarity distance. The L^*_{MDP} model is larger than the IoAlergia model as in previous experiments, but in this example it is smaller than the true model. This happens because many states are never reached during learning, as reaching them within a bounded number of steps has a very low probability; see, for instance, the fifth model-checking query. This query determines the maximum probability of finishing the protocol within less than 40 steps, but without consensus, as process p_1 chooses heads and process p_2 chooses tails. The computed probability of this event is very low.

The other model-checking queries consider the maximum probability of: (1) finishing the protocol without consensus, (2) finishing the protocol with consensus, (3) finishing the protocol without reaching an intermediate counter state of 5, (4) finishing the protocol without reaching an intermediate counter state of 4. The queries (5) to (8) bound the number of steps by less than 40, but consider the same properties as the queries (1) to (4), which are unbounded. Here, we see that the model-checking results computed with the IoAlergia model are more accurate in some cases, but L^*_{MDP} produces more accurate

Table 7.5: Results for learning models of the slot machine with $t_{\text{unamb}} = 0.9$

	true	L^*_{MDP}	IoAlergia
# outputs	-	$4{,}752{,}687$	$4{,}752{,}691$
# traces	-	$1{,}567{,}487$	$594{,}086$
time [s]	-	$3{,}381.0$	60.3
# states	109	109	86
$\delta_{0.9}$	-	0.1632	0.2983
$\mathbb{P}_{\max}(F(\text{Pr10}))$	0.3637	0.3769	0.4169
$\mathbb{P}_{\max}(F(\text{Pr2}))$	0.6442	0.6697	0.6945
$\mathbb{P}_{\max}(F(\text{Pr0}))$	1.0000	1.0000	1.0000
$\mathbb{P}_{\max}(X(X(\text{bar} - \text{bar} - \text{blank})))$	0.1600	0.1615	0.1639
$\mathbb{P}_{\max}(X(X(X(\text{apple} - \text{bar} - \text{bar}))))$	0.2862	0.2865	0.2776
$\mathbb{P}_{\max}(\neg(F^{<10}(\text{end})))$	0.2500	0.3013	0.3283
$\mathbb{P}_{\max}(X(X(X(\text{apple} - \text{apple} - \text{apple}))) \wedge (F(\text{Pr0})))$	0.0256	0.0262	0.0107

Table 7.6: Results for learning models of the slot machine with $t_{\text{unamb}} = 0.99$

	true	L^*_{MDP}	IoAlergia
# outputs	-	$24{,}290{,}643$	$24{,}282{,}985$
# traces	-	$7{,}542{,}332$	$3{,}036{,}332$
time [s]	-	$18{,}048.0$	518.9
# states	109	109	97
$\delta_{0.9}$	-	0.0486	0.2518
$\mathbb{P}_{\max}(F(\text{Pr10}))$	0.3637	0.3722	0.3991
$\mathbb{P}_{\max}(F(\text{Pr2}))$	0.6442	0.6552	0.6997
$\mathbb{P}_{\max}(F(\text{Pr0}))$	1.0000	1.0000	1.0000
$\mathbb{P}_{\max}(X(X(\text{bar} - \text{bar} - \text{blank})))$	0.1600	0.1607	0.1597
$\mathbb{P}_{\max}(X(X(X(\text{apple} - \text{bar} - \text{bar}))))$	0.2862	0.2866	0.2851
$\mathbb{P}_{\max}(\neg(F^{<10}(\text{end})))$	0.2500	0.2606	0.4000
$\mathbb{P}_{\max}(X(X(X(\text{apple} - \text{apple} - \text{apple}))) \wedge (F(\text{Pr0})))$	0.0256	0.0264	0.0128

results overall. The absolute difference from the true values averaged over all model-checking results is about 0.066 for L^*_{MDP}, approximately half of IoAlergia's average absolute difference of 0.138.

We see an increase in runtime compared to the gridworld examples. Learning the consensus-protocol model took about 53 minutes, whereas learning the gridworld models took less than 3 minutes. This is caused by the larger state space of the consensus protocol, since the precomputation time for equivalence testing grows with the state space. Overall, we see a similar picture as before: L^*_{MDP} trades runtime for increased accuracy which we deem reasonable.

7.6.5 Slot-Machine Experiments

The final experiments in this section are based on the slot machine example that we also used in Section 6.3 and which we adapted from Mao et al. [198, 199]. Here, we set the maximum number of rounds m to 3, leading to a state space of 109 states for the minimal true model of the slot machine.

We configured sampling for IoAlergia with $p_l = 0.125$ and we set the following parameters for L^*_{MDP}: $n_{\text{resample}} = n_{\text{retest}} = n_{\text{test}} = 300$, $p_{\text{stop}} = 0.25$, $p_{\text{rand}} = 0.25$, $r_{\min} = 500$ and $r_{\max} = 20000$. To demonstrate the influence of the parameter t_{unamb}, we performed experiments with $t_{\text{unamb}} = 0.9$ and with $t_{\text{unamb}} = 0.99$.

Table 7.5 and Table 7.6 show the results for $t_{\text{unamb}} = 0.9$ and $t_{\text{unamb}} = 0.99$, respectively. Configured with $t_{\text{unamb}} = 0.9$, L^*_{MDP} stopped after 2,988 rounds and it stopped after 12,879 rounds if configured with $t_{\text{unamb}} = 0.99$. We see here that learning an accurate model of the slot machine requires a large amount of samples; in the case of $t_{\text{unamb}} = 0.99$, we sampled 7,542,332 traces containing 24,290,643 outputs. The amount of outputs is almost 10 times as high as for the gridworld examples. However, we also see that sampling more traces clearly pays off. The L^*_{MDP} results shown in Table 7.6 are much better than those shown in Table 7.5. Notably the state space stayed the same. Thus, the model learned with fewer traces presumably includes some incorrect transitions. This is exactly what our stopping heuristic aims to avoid; it aims to avoid ambiguous membership of traces in compatibility classes to reduce the uncertainty in creating transitions.

Like in the gridworld experiments, the L^*_{MDP} models have the same size as the true model, while the IoAlergia models are smaller. We also see in both settings for L^*_{MDP} that L^*_{MDP} models are more accurate than IoAlergia models, with respect to bisimilarity distance and with respect to model-checking results. While the experiment with $t_{\text{unamb}} = 0.99$ required the most samples among all experiments, it also led to the lowest bisimilarity distance.

The first three model-checking queries determine the maximum probability of reaching one of the three prizes issued by the slot machine. The next two queries consider certain reel configurations reached after exactly two and three steps, respectively. The sixth query checks the maximum probability that ending the game can be avoided for 9 steps. The final query computes the maximum probability of reaching the prize Pr0 and observing the reel configuration $\text{apple} - \text{apple} - \text{apple}$ after exactly three steps. It is noteworthy that the model-checking results for the L^*_{MDP} model learned with $t_{\text{unamb}} = 0.99$ are within a low range of approximately 0.01 of the true results.

A drawback of L^*_{MDP} compared to IoAlergia is again the learning runtime, as L^*_{MDP} with $t_{\text{unamb}} = 0.99$ required about 5 hours while learning with IoAlergia took only about 8.7 minutes. As pointed out several times, in a non-simulated environment, the sampling time would be much larger than 5 hours, such that the learning runtime becomes negligible. Consider, for instance, a scenario where the sampling of a single trace takes 20 milliseconds. The sampling time of L^*_{MDP} is about 42 hours in that scenario which is about 8.4 times as long as the learning runtime.

7.6.6 Discussion and Threats to Validity

Our case studies demonstrated that L^*_{MDP} is able to achieve better accuracy than IoAlergia. The bisimilarity distances of L^*_{MDP} models to the true models were generally lower and the model checking results were more accurate. These observations should be investigated in further case studies. It should be noted, though that the considered systems have different characteristics. The gridworld has small state space, but it is strongly connected and the different terrains lead to different probabilistic decisions. For instance, if we try to enter *mud* there is a probability of 0.4 of entering one of the neighbouring tiles, whereas entering *concrete* is generally successful (the probability of entering other tiles instead is 0). The consensus protocol has a large state space with many different outputs and finishing the protocol takes at least 14 steps. The slot machines requires states to be distinguished based on subtle differences in probabilities, because the probability of seeing bar decreases in each round.

Due to the high runtimes of L^*_{MDP} and the bisimilarity-distance computation, we did not perform repeated experiments. Hence, there is a risk that the accuracy of learned models varies greatly.

L^*_{MDP} has several parameters that affect performance and accuracy. We plan to investigate the influence of parameters in further experiments. For the presented experiments, we fixed most of the parameters except for n_{retest}, n_{test} and n_{resample}. However, we observed that results are robust with respect to these parameters. We increased, for instance, n_{resample} from 300 for the first gridworld to 1,000 for the second gridworld. Both settings led to approximately the same results, as learning simply performed fewer rounds with $n_{\text{resample}} = 1{,}000$. Nevertheless, we will examine in further experiments if the fixed parameters are indeed appropriately chosen and if guidelines for choosing other parameters can be provided.

L^*_{MDP} and IOALERGIA learn from different traces, thus the trace selection may actually be the main reason for the better accuracy of L^*_{MDP}. We examined if this is the case by learning IOALERGIA models from two types of given traces: traces with uniform input selection and traces sampled during learning with L^*_{MDP}. We noticed that models learned from L^*_{MDP} traces led to less accurate results overall, especially in terms of bisimilarity distance. Therefore, we reported only results for models learned with IOALERGIA from traces with uniformly distributed inputs.

7.7 Summary

In this chapter, we presented two L^*-based algorithms for learning MDPs. For our exact learning algorithm $L^*_{\text{MDP}^e}$, we assume an ideal setting that allows to query information about the SUL with exact precision. Subsequently, we relaxed our assumptions by approximating exact queries through sampling SUL traces via directed testing. These traces serve to infer the structure of hypothesis MDPs, to estimate transition probabilities and to check for equivalence between the SUL and learned hypotheses. The resulting sampling-based L^*_{MDP} iteratively learns approximate MDPs which converge to the correct MDP in the large sample limit. We implemented L^*_{MDP} and evaluated it by comparing it to IOALERGIA [199], a state-of-the-art passive learning algorithm for MDPs. The evaluation showed that L^*_{MDP} is able to produce more accurate models. To the best of our knowledge, L^*_{MDP} is the first L^*-based algorithm for MDPs that can be implemented via testing. Experimental results and the implementation can be found in the evaluation material for L^*_{MDP} [261].

7.8 Results and Findings

In the following, we want to discuss the results of our experiments with L^*_{MDP} and our findings concerning learning of MDPs. The research questions that are relevant in this context shall provide a framework for the discussion.

RQ 1.1 Are randomised testing techniques a sensible choice for learning-based testing? As in Chapter 6, we successfully applied randomised online conformance testing in automata learning. Furthermore, we were able to adapt a testing technique, the *transition coverage* testing strategy proposed for deterministic systems in Chapter 4, to state-coverage-based testing in stochastic automata learning. The adapted testing technique enabled us to reliably learn models that are close to the true model of the SUL. Hence, we can provide a positive answer to **RQ 1.3** in this context as well, which addresses whether correct models can be learned reliably. As discussed in Section 7.1, we do not target absolute correctness in the presence of stochastic behaviour.

RQ 1.2 What guarantees can be given if randomised testing is applied? In the context of MDP learning, we need to take the stochastic nature of the SUL into account. The guarantee we can provide for L^*_{MDP} is that it learns correct models in the limit. Mao et al. [199] provide this guarantee for IOALERGIA as well. They also state that PAC learnability results [276], which are stronger, are difficult to achieve in a verification context, since appropriate distance measures are difficult to define.

RQ 2.2 When can we stop learning in the presence of uncertain behaviour? As mentioned above, we learn correct models in the limit, but we do not provide stronger convergence results. For this reason, we cannot compute the sample size necessary to achieve a given model accuracy. However, we developed a stopping heuristic that addresses uncertainties in the hypothesis construction which is based on statistics. We empirically demonstrated that this heuristic works well by learning accurate models with L^*_{MDP}. In the slot machine example in Section 7.6.5, we specifically examined the influence of this heuristic. By setting the parameter t_{unamb} to a larger value, demanding a higher degree of certainty before stopping, we improved the model accuracy substantially.

RQ 2.3 Is test-based active learning feasible in the presence of stochastic behaviour? Since we learned accurate models with L^*_{MDP}, we can answer this question positively. We discovered various pitfalls in the development of L^*_{MDP}, though. These pitfalls should be avoided or at least considered in the development of a test-based L^*-style algorithm for stochastic systems.

- *Observation table size:* the size of observation tables should be controlled in our experience. This is the reason why we remove rows from observation tables.

  Incorrect statistical test results may cause rows to be added that are not needed. This is likely to occur during learning and with increasing table size, the likelihood of such events even increases. A larger table contains more data, which leads to more statistical tests and consequently to more opportunities for incorrect decisions. Hence, a growing observation table may cause additional growth in a snowball effect.

  The initial version of L^*_{MDP} did not remove rows, i.e. it did not apply the TRIM function in Algorithm 7.4. This version successfully learned the coffee machine MDP shown in Example 5.1, but it could not learn gridworld MDPs. In preliminary experiments with gridworld models, the observation tables grew to a size that prevented learning from terminating in reasonable time, due to the runtime of operations like closedness checks.

- *Unnecessary counterexamples:* equivalence queries return $none$ if the chaos state is reachable in the current hypothesis. This is motivated by the fact that we trivially know that the hypothesis is incorrect in such cases. Rather than adding new traces to the observation table, the information stored in the observation table should be refined.

Concluding Remarks. The evaluation of L^*_{MDP} showed promising results. We applied it to successfully learn a protocol model, the shared coin consensus protocol, for instance. Despite its hardness, it would be worthwhile to analyse L^*_{MDP} with respect to PAC learnability [276] to provide stronger convergence guarantees. L^*_{MDP} also provides room for experimentation. Different testing techniques, such as reachability-property-directed sampling presented in Chapter 6, could be applied in equivalence queries.

8 Learning Timed Automata via Genetic Programming

Declaration of Sources

We presented the work discussed in this chapter at the FORMATS 2019. The chapter is mainly based on the conference paper that appeared in the proceedings of the FORMATS 2019 [267] and on a technical report available via `arxiv.org` [264], which includes additional technical details.

8.1 Introduction

In the exploratory research discussed in Section 3.6, we identified the need to capture time-dependent behaviour in learned models. Otherwise, certain aspects of networked systems cannot be adequately covered by learning-based testing. In the MQTT case study, we had to abstract away from timing details, by employing a timeout mechanism; see Section 3.3.

Time-dependent behaviour is often modelled with timed automata (TA) [32]. These are finite automata extended with real-valued variables, called clocks, that measure the progress of time. Clocks can be reset and based on clock valuations, actions can be enabled or disabled. Learning of TA models has received little attention, with two notable exceptions that we will discuss briefly. Verwer et al. presented passive learning algorithms for deterministic *real-time automata* and a probabilistic variant thereof [279, 280]. Like IoAlergia [199], the passive algorithm that we used to learn MDPs (see Sections 5.4 and 6.2), these algorithms are based on state merging. In a different line of research, Grinchtein et al. proposed active learning algorithms for deterministic *event-recording automata* [125, 126]. They extended Angluin's L^* [37] in one approach [126] and used *timed decision trees* as a basis in another work [125]. More recently, Jonsson and Vaandrager developed an L^*-based technique for Mealy machines with timers [157] that can model timeouts accurately.

Limitations of the State of the Art. The existing learning solutions for TA contain several limitations. Real-time automata learned by Verwer et al. [279, 280] are restricted to one clock, which is reset on every transition. Thus, these automata allow modelling delays depending on the current location, but they cannot model more complex conditions on timing. Event-recording automata learned by Grinchtein et al. [125, 126] are also less expressive than TA [34]. These automata contain one clock for each action that is always reset when the corresponding action is performed. Moreover, the learning algorithms for this class of automata have a high runtime complexity. Depending on the specific subclass of event-recording automata that should be learned, the query complexity can be double exponential in the alphabet size [126]. Both real-time automata and event-recording automata do not distinguish between inputs and outputs, as DFA. Introducing such a distinction is not as simple as partitioning the set of actions into two disjoint sets. Adapting either of the existing learning approaches would require substantial changes. Since this thesis takes a testing view, we generally consider input-enabled systems, as is common in testing [64, 137, 154, 170, 270, 271]. Mealy machines and MDPs that served as modelling formalisms in previous chapters are both input enabled. As for discrete-time systems, different rules apply to inputs of real-time systems than to their outputs, since we can control inputs, while outputs are produced by the systems.

Goal. We present a learning approach for TA in this chapter. With this approach, we want to overcome limitations of existing techniques. It shall be possible to learn input-enabled TA with arbitrary clock resets from observations of a real-time SUL. Such observations may, for instance, be recorded during testing. Arbitrary clock resets enable modelling of complex time-dependent interdependencies between actions. Therefore, we aim at learning general TA, as used by the model checker UPPAAL [178]. However, these automata usually do not have canonical forms [126] which complicates the development of learning techniques. Due to that and because we cannot benefit from assumptions about clock resets like existing techniques [125, 126, 279, 280], we follow a non-traditional automata learning approach. We apply genetic programming on TA to basically search for TA that are consistent with given SUL observations. This is also motivated by the successful application of genetic programming in program synthesis by Katz and Peled [159]. In contrast to previous chapters, we present a passive technique as a first step towards learning TA. However, we consider a test-based setting, since we learn from test observations and we target models suitable in this setting. This serves the goal of enabling an active extension of the basic passive technique. We will actually present an active extension that has been developed and evaluated by Andrea Pferscher in her master's thesis [234] in Chapter 9.

Assumptions. Despite targeting general TA, we need to place some assumptions on the SULs. This is motivated by a classic result of language learnability by Gold [122]. He showed that only classes of finite formal languages can be learned solely from words of the languages. Basically, this means that for classes of infinite languages it is not sufficient to have only positive examples (e.g. system traces), but also additional information is required for learnability. Since TA are generally able to produce an infinite number of different traces, we need to place some restrictions on allowed SUL behaviour to avoid tackling an ill-posed problem. Verwer et al. [279], for instance, learn deterministic real-time automata from positive and negative samples. Negative samples are sets of traces that *cannot* be produced by the SUL. In Mealy machine learning, we assume determinism, thus any trace introducing non-determinism is a negative example.

Therefore, we also consider SULs to be deterministic. More concretely, we assume three different forms of determinism, one is related to discrete behaviour and two are related to time-dependent behaviour. The restrictions required by our approach are that the SULs are *deterministic*, *output urgent*, and with *isolated outputs* (a special form of output determinism) [137]. *Determinism* is defined similarly to determinism of finite automata. *Output urgency* means that outputs shall be produced as soon as possible. A TA has *isolated outputs* if at most one output is enabled at any time.

These assumptions often hold in applications. While specifications are generally vague, especially with respect to timing, and leave freedom to the actual implementation, implementations are often concrete instantiations of a design. They can be assumed to implement specific choices deterministically with the goal of satisfying a specification [92]. Thus, we do not consider the restriction to deterministic behaviour as a severe limitation, since we learn from concrete implementations. Moreover, determinism is a common assumption in automata learning.

Scope and Contribution. We use a form of genetic programming [167] to passively learn a deterministic TA consistent with a given set of timed traces which are sampled through random testing. Put differently, we perform a metaheuristic search to find a TA that produces the same traces as the SUL. The implementation of this learning technique is available for download [262]. It contains a graphical user interface for demonstration purposes. We evaluate the technique on four manually created TA and several randomly generated TA. Our evaluation demonstrates that the search reliably converges to a TA consistent with test cases given as training data. Furthermore, we simulate learned TA on independently produced test data to show that our identified solutions generalise well and that they do not overfit to the training data.

Chapter Structure. The rest of this chapter is structured as follows. Section 8.2.1 contains background information on TA and genetic programming. Section 8.3 describes our approach to learning TA. The evaluation of this approach is presented in Section 8.4. In Section 8.5, we provide a summary and conclude with a discussion of our findings and results in Section 8.6.

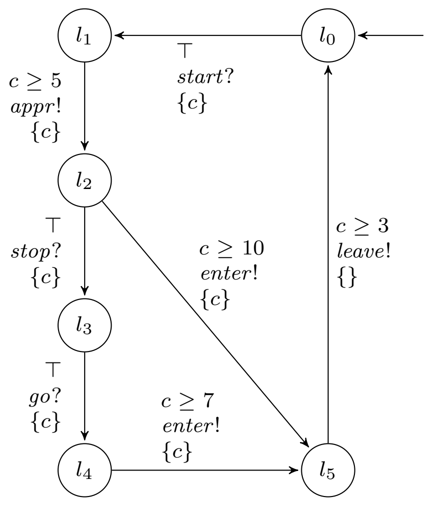
8.2 Preliminaries

8.2.1 Timed Automata

Timed automata are finite automata enriched with real-valued variables called clocks [32]. Clocks measure the progress of time which elapses while an automaton resides in some location. Transitions can be constrained based on clock values and clocks may be reset on transitions. We denote the set of clocks by $\mathcal{C}$ and the set of guards over $\mathcal{C}$ by $\mathcal{G}(\mathcal{C})$. Guards are conjunctions of constraints of the form $c \oplus k$, with $c \in \mathcal{C}, \oplus \in \{>, \geq, \leq, <\}, k \in \mathbb{N}$. Edges of TA are labelled by input and output actions, denoted by Σ_I and Σ_O respectively, with $\Sigma = \Sigma_I \cup \Sigma_O$ and $\Sigma_I \cap \Sigma_O = \emptyset$. Input labels are suffixed by '?' and output labels end with '!'. A TA over $(\mathcal{C}, \Sigma)$ is a triple $\langle L, l_0, E\rangle$, where L is a finite non-empty set of locations, $l_0 \in L$ is the initial location and E is the set of edges, with $E \subseteq L \times \Sigma \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$. We write $l \xrightarrow{g,a,r} l'$ for an edge $(l, g, a, r, l') \in E$ with guard g, action label a, and clock resets r.

Example 8.1 (Train TA Model). Figure 8.1 shows a TA model of a train approaching, entering, and leaving a track section. The TA is defined by the inputs $\Sigma_I = \{\mathit{start}?, \mathit{stop}?, \mathit{go}?\}$, the outputs $\Sigma_O = \{\mathit{appr}!, \mathit{enter}!, \mathit{leave}!\}$, the clock $\mathcal{C} = \{c\}$, the locations $L = \{l_0, \ldots, l_5\}$, and the edges $E = \{l_0 \xrightarrow{\top,\mathit{start}?,\{c\}} l_1, l_1 \xrightarrow{c\geq 5,\mathit{appr}!,\{c\}} l_2, l_2 \xrightarrow{\top,\mathit{stop}?,\{c\}} l_3, l_2 \xrightarrow{c\geq 10,\mathit{enter}!,\{c\}} l_5, l_3 \xrightarrow{\top,\mathit{go}?,\{c\}} l_4, l_4 \xrightarrow{c\geq 7,\mathit{enter}!,\{c\}} l_5, l_5 \xrightarrow{c\geq 3,\mathit{leave}!,\{\}} l_0\}\}$. From the initial location l_0, the train accepts the input $\mathit{start}?$, resetting clock c, denoted by c in curly braces. After that, it can produce the output $\mathit{appr}!$ if $c \geq 5$, thus the train may approach 5 time units after it is started.

Semantics. The semantics of a TA is given by a timed transition system (TTS) $\langle Q, q_0, \Sigma, T\rangle$, with states $Q = L \times \mathbb{R}_{\geq 0}{}^{\mathcal{C}}$, initial state q_0, and transitions $T \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$, for which we write $q \xrightarrow{e} q'$ if $(q, e, q') \in T$. A state $q = (l, \nu)$ is a pair consisting of a location l and a clock valuation ν. For $r \subseteq \mathcal{C}$, we denote resetting the clocks in r to zero by $\nu[r]$, i.e. $\forall c \in r : \nu[r](c) = 0$ and $\forall c \in \mathcal{C} \setminus r : \nu[r](c) = \nu(c)$. Let $(\nu + d)(c) = \nu(c) + d$ for $d \in \mathbb{R}_{\geq 0}, c \in \mathcal{C}$ denote the progress of time and $\nu \models \phi$ denote that valuation ν satisfies formula ϕ. Finally, $\mathbf{0}$ is the valuation assigning zero to all clocks, which we use to define the initial state q_0 as $(l_0, \mathbf{0})$. Transitions of TTSs are either delay transitions $(l, \nu) \xrightarrow{d} (l, \nu + d)$ for a delay $d \in \mathbb{R}_{\geq 0}$, or discrete transitions $(l, \nu) \xrightarrow{a} (l', \nu[r])$ for an edge $l \xrightarrow{g,a,r} l'$ such that $\nu \models g$. Delays are usually further constrained, for instance, by invariants limiting the sojourn time in locations [137].



l_0
l_1
l_2
l_3
l_4
l_5
$\top$ $start?$ $\{c\}$
$c \geq 5$ $appr!$ $\{c\}$
$\top$ $stop?$ $\{c\}$
$c \geq 10$ $enter!$ $\{c\}$
$c \geq 3$ $leave!$ $\{\}$
$\top$ $go?$ $\{c\}$
$c \geq 7$ $enter!$ $\{c\}$

Figure 8.1: A TA modelling a passing train

Timed Traces. We use the terms timed traces and test sequences similarly to Springintveld et al. [256]. The latter are sequences of inputs and corresponding execution times, while the former are sequences of inputs and outputs, together with their times of occurrence. Timed traces are produced in response to test sequences. A test sequence ts is an alternating sequence of non-decreasing time stamps t_j and inputs i_j, that is, $ts = t_1 \cdot i_1 \cdots t_n \cdot i_n \in (\mathbb{R}_{\geq 0} \times \Sigma_I)^*$ with $\forall j \in \{1, \ldots, n-1\} : t_j \leq t_{j+1}$. Informally, a test sequence prescribes that i_j should be executed at time t_j. A timed trace $tt \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ consists of inputs interleaved with outputs produced by a timed system. Analogously to test sequences, timestamps in timed traces are non-decreasing.

Assumptions on Timed Systems. Testing based on TA often places further assumptions on TA [137, 256]. Since we consider a test-based setting and we learn models from test observation we make similar assumptions to enable learning (closely following Hessel et al. [137]). We describe these assumptions on the level of semantics and use $q \xrightarrow{a}$ to denote $\exists q' : q \xrightarrow{a} q'$ and $q \not\xrightarrow{a}$ for $\nexists q' : q \xrightarrow{a} q'$:

1. *Determinism.* A TA is deterministic iff for every state $q = (l, \nu)$ and every action $a \in \Sigma$, whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' = q''$.
2. *Input Enabledness.* A TA is input enabled iff for every state $q = (l, \nu)$ and every input $i \in \Sigma_I$, we have $q \xrightarrow{i}$.
3. *Output Urgency.* A TA shows output-urgent behaviour if outputs occur immediately as soon as they are enabled, i.e. for $o \in \Sigma_O$, if $q \xrightarrow{o}$ then $q \not\xrightarrow{d}$ for all $d \in \mathbb{R}_{>0}$. Thus, outputs must not be delayed.
4. *Isolated Outputs.* A TA has isolated outputs iff whenever an output may be executed, then no other output is enabled, i.e. if $\forall o \in \Sigma_O, \forall o' \in \Sigma_O : q \xrightarrow{o}$ and $q \xrightarrow{o'}$ implies $o = o'$.

It is necessary to place restrictions on the sojourn time in locations to establish output urgency. Deadlines provide a simple way to model the assumption that systems are output urgent [65]. With deadlines it is possible to model eager actions. We use this concept and implicitly assume all learned output edges to be eager. This means that outputs must be produced as soon as their guards are satisfied. For that, we extend the semantics given above by adding the following restriction:

$$\text{delays } (l, \nu) \xrightarrow{d} (l', \nu + d) \text{ are only possible if}$$
$$\forall\, d' \in \mathbb{R}_{\geq 0}, d' < d : \nu + d' \models \neg \bigvee_{g \in G_O} g,$$
$$\text{where } G_O = \{g \mid \exists l', a, r : l \xrightarrow{g,a,r} l', a \in \Sigma_O\} \text{ are the guards of outputs in location } l.$$

To avoid issues related to the exact time at which outputs should be produced, we further restrict the syntax of TA by disallowing strict lower bounds for output edges. The model checker UPPAAL [178] uses invariants rather than deadlines to limit sojourn time. In order to analyse TA using UPPAAL, we use the translation from TA with deadlines to TA with invariants presented by Gómez [123]. Additionally, we apply two slight adaptations related to inputs and the properties discussed above. Whenever inputs and outputs are enabled, we disable inputs implicitly, i.e. we restrict the semantics further by adding a transition $(l, \nu) \xrightarrow{i} (l', \nu[r])$ for $i \in \Sigma_I$ only if $\forall l \xrightarrow{g,a,r} l' : a \in \Sigma_O \rightarrow \nu \not\models g$. This follows the reasoning that we cannot block outputs and that outputs occur urgently. Furthermore, we implicitly add self-loop transitions to all states $s = (l, \nu)$ for inputs i undefined in s,i.e. we add $(l, \nu) \xrightarrow{i} (l, \nu)$ if $\nu \not\models \bigvee_{\exists l', r : l \xrightarrow{g,i,r} l'} g$ and $\forall o \in \Sigma_O : s \overset{o}{\not\rightarrow}$ (due to isolated outputs). This ensures input enabledness while avoiding TA cluttered with input self-loops. It also allows to ignore input enabledness during genetic programming. As a result, mutations may remove any input edge without restrictions.

Testability and Learnability. The above assumptions placed on SULs ensure testability [137]. Assuming that SULs can be modelled in some modelling formalism is usually referred to as *testing hypothesis* [269]. Placing the same assumptions on learned models simplifies checking conformance between model and SUL.

Under these assumptions, the execution of a test sequence uniquely determines a response in the form of a timed trace [256] and due to input enabledness we may execute any test sequence. Therefore, we can view a timed SUL satisfying our assumptions as a function $t : (\mathbb{R}_{\geq 0} \times \Sigma_i)^* \rightarrow (\mathbb{R}_{\geq 0} \times \Sigma)^*$, mapping from test sequences to timed traces. Based on that, we can specify how to derive *negative* samples, even though the observed timed traces produced by the SUL are *positive* samples. If we observe the timed trace $tt = t(ts)$ in response to performing the test sequences ts on the SUL, then any tt' which is not a prefix of tt is a negative sample trace. This means that correct models must not produce tt'.

This form of determinism allows us to use equivalence as conformance relation between learned models and the SUL. What is more, we can approximate checking equivalence between the learned models and the SUL by simulating test sequences on the models and checking for equivalence between the SUL's responses and the responses predicted by the models.

8.2.2 Genetic Programming

Genetic programming [167] is a search-based technique to automatically generate programs exhibiting some desired behaviour. Similar to *Genetic Algorithms* [209], it is inspired by nature. Programs, also called individuals, are iteratively refined by: (1) fitness-based selection followed by (2) operations altering program structure, such as mutation and crossover. Fitness measures the quality of individuals in a problem-specific way, for instance, based on tests. In this case, one could assign a fitness value proportional to the number of tests that are passed by an individual. The following basic functioning principle underlies genetic programming.

1. Randomly create an initial population of individuals.

2. Evaluate the fitness of each individual in the population.

3. If an acceptable solution has been found or the maximum number of iterations has been performed: **stop** and output the best individual

4. Otherwise repeatedly select an individual based on fitness and apply one of:
   - *Mutation:* change a part of the individual to create a new individual.
   - *Crossover:* select another individual according to its fitness and combine both individuals to create offspring.
   - *Reproduction:* copy the individual to create a new equivalent individual.

5. Form a new population from the new individuals and go to Step 2.

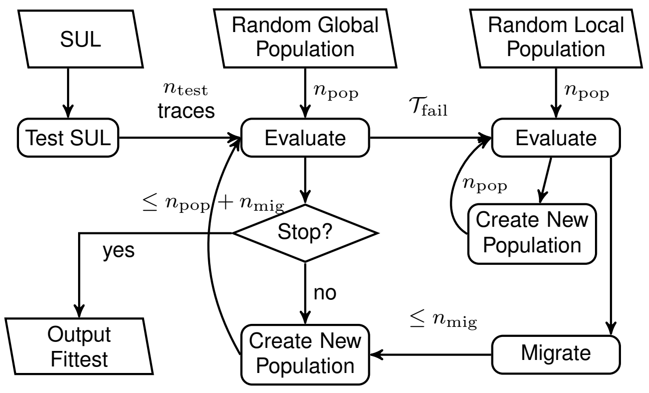


Figure 8.2: Overview of genetic programming for timed automata

Several variations and extensions of this general approach exist. In the following, we are going to discuss additional details regarding extensions that will be applied in Section 8.3.

Mutation Strength and Parameter Adaptation. In metaheuristic search techniques, such as evolution strategies [59], mutation strength typically describes the level of change caused by mutations. This parameter heavily influences search, therefore there exist various schemes to adapt it. Since the optimal parameter value is problem-specific, it makes sense to evolve it throughout the search together with the actual individuals. Put differently, individuals are equipped with their own mutation strength, which is mutated during the search. Individuals with good mutation strength are assumed to survive fitness-based selection and to produce offspring.

Elitism. Elitist strategies keep track of a portion of the fittest individuals found so far and copy them into the next generation [209]. This may improve performance, as correctly identified partial solutions cannot be forgotten.

Subpopulations and Migration. Due to their nature, genetic algorithms and genetic programming lend themselves to parallelisation. Several populations may, for instance, be evolved in parallel, which is particularly useful if speciation is applied [222]. In speciation, different subpopulations explore different parts of the search space. In order to exchange information between the subpopulations, it is common that individuals migrate between them.

8.3 Genetic Programming for Timed Automata

8.3.1 Overview

In this section, we discuss our implementation of genetic programming. Figure 8.2 provides an overview of the performed steps, while Figure 8.3 shows the creation of a new population in more detail.

We first test the SUL by generating and executing n_{test} test sequences to sample n_{test} timed traces. Our goal is then to genetically program a TA consistent with the sampled timed traces. Put differently, we want to generate a TA that produces the same outputs as the SUL in response to the inputs of the test sequences. For the following discussion, we say that a TA passes a timed trace t of the SUL if it produces the same trace t when simulating the test sequence corresponding to t. Otherwise it fails t. In addition to passing all timed traces, the final TA shall be deterministic. This is achieved by assigning larger fitness values to deterministic solutions. Both mutation and crossover can create non-deterministic intermediate solutions, which might help the search in the short-term and will be resolved in future generations.

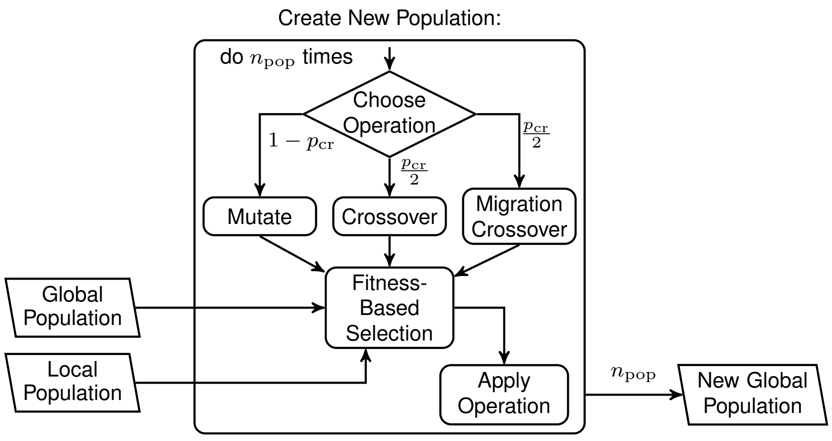


Figure 8.3: Creating a new global population of timed automata

Generally, we evolve two populations of TA simultaneously, a global population, evaluated on all the traces, and a local population, evaluated only on the traces that fail on the fittest automaton of the global population. Both are initially created equally and contain n_{pop} TA. After initial creation, the global population is evaluated on all n_{test} traces. During that, we basically test the TA to check how many traces each TA passes and assign fitness values accordingly; i.e., the more passed traces, the fitter. Additionally, we add a fitness penalty for model size.

The local population is evaluated only on a subset $\mathcal{T}_{\mathrm{fail}}$ of the traces. This subset $\mathcal{T}_{\mathrm{fail}}$ contains all traces which the fittest TA fails, and which likely most of the other TA fail as well. With the local population, we are able to explore new parts of the search space more easily since the local search may ignore functionality already modelled by the global population. We integrate functionality found via this local search into the global population through migration and migration combined with crossover. To avoid overfitting to a low number of traces, we ensure that $\mathcal{T}_{\mathrm{fail}}$ contains at least $\frac{n_{\mathrm{test}}}{100}$ traces. If there are fewer actually failing traces, we add randomly chosen traces from all n_{test} traces to $\mathcal{T}_{\mathrm{fail}}$.

After evaluation, we stop if we either reached the maximum number of generations g_{max} or the fittest TA passes all traces and has not changed in g_{change} generations. Note that two TA passing all traces may have different fitness values depending on model size, thus g_{change} controls how long we try to decrease the size of the fittest TA. The motivation for this is that smaller TA are less complex and simpler to comprehend. Additionally, they can be expected to be more general than larger TA and less likely to overfit to the given data.

If the search has not stopped, we create new populations of TA, which works slightly differently for the local and the global population. Figure 8.3 illustrates the creation of a new global population. Before creating new TA, existing TA may migrate from the local to the global population. For that, we check each of the fittest n_{mig} local TA and add it to the global population, if it passes at least one trace from $\mathcal{T}_{\mathrm{fail}}$. We generally set n_{mig} to $\frac{5n_{\mathrm{pop}}}{100}$, i.e., the top five per cent of the local population are allowed to migrate. After migration we create n_{pop} new TA through the application of one of three operations:

1. *with probability* $1 - p_{\mathrm{cr}}$*:* mutation of a TA from the global population
2. *with probability* $\frac{p_{\mathrm{cr}}}{2}$*:* crossover of two TA from the global population
3. *with probability* $\frac{p_{\mathrm{cr}}}{2}$*:* crossover of two TA, one from each population

The rationale behind migration combined with crossover is that migrated TA may have low fitness from a global point of view and will therefore not survive selection. They may, however, have desirable features which can be transferred via crossover. For the local population, we perform the same steps, but without migration from the global population, in order to keep the local search independent. Once we have new populations, we start a new generation by evaluating the new TA.

Table 8.1: Parameters for the initial creation of TA

Name	Short description
n_{pop}	number of TA in population
Σ_I & Σ_O	the input and output action labels on edges
n_{clock}	number of clocks in the set of clocks $\mathcal{C}$
$c_{\max}$	approximate largest constant in clock constraints

A detail not illustrated in Figure 8.2 is our implementation of *elitism* [209]. We always keep track of the fittest TA found so far for both populations. In each generation, we add these fit TA to their respective populations after performing mutation and crossover.

Parameters. The genetic-programming implementation can be controlled by a large number of parameters. To ease applicability and to avoid the need for meta-optimisation of parameter settings for a particular SUL, we fixed as many as possible to constant values. The actual values, such as $\frac{5n_{\text{pop}}}{100}$ for n_{mig}, are motivated by experiments. The remaining parameters can usually be set to default values or chosen based on guidelines. For instance, n_{pop}, $g_{\max}$, and n_{test} may be chosen as large as possible, given available memory and maximum computation time. The same holds for g_{change}.

8.3.2 Creation of Initial Random Population

We initially create n_{pop} random TA parameterised by: (1) the labels Σ_I and Σ_O, (2) the number of clocks n_{clock}, and (3) the approximately largest constant in clock constraints $c_{\max}$. These parameters are also summarised in Table 8.1. Note that $c_{\max}$ is an approximation, because mutations may increase constants to values larger than $c_{\max}$, but with low probability. Each TA has initially only two locations, as we intend to increase size and with that complexity only through mutation and crossover. Moreover, TA are assigned the given action labels and have n_{clock} clocks. During creation we add random edges, such that at least one edge connects the initial location to the other location. These edges are entirely random, where the number of constraints in guards as well as the number of clock resets are geometrically distributed with fixed parameters. The edge labels, the relational operators and constants in constraints are chosen uniformly at random from the respective sets Σ, $\{<, \leq, \geq, >\}$, and $[0 \,..\, c_{\max}]$ (operators for outputs exclude $>$). The source and target locations are also chosen uniformly at random from the set of locations, i.e. initial choices consider two locations.

If the required number of clocks is not known a priori, we suggest setting $n_{\text{clock}} = 1$ and increasing this value only if it is not possible to find a valid TA. A similar approach can be used for $c_{\max}$.

8.3.3 Fitness Evaluation

Simulation. We simulate timed traces on TA to evaluate their fitness. At the beginning of this section, we mentioned failing and passing traces, but the evaluation is actually more fine grained. We execute the inputs of each timed trace and observe the produced outputs until (1) the simulation is complete, (2) an expected output is not observed, or (3) output isolation is violated (output non-determinism).

As pointed out in Section 8.2.1, if $\mathcal{T}$ is a deterministic, input-enabled TA with isolated and urgent outputs and ts is a test sequence, then executing ts on $\mathcal{T}$ uniquely determines a timed trace tt [256]. By the testing hypothesis, the SUL fulfils these assumptions. However, TA generated through mutation and crossover are input-enabled, but they may show non-deterministic behaviour. Hence, simulating a test sequence or a timed trace on a generated TA may follow multiple paths of states. Some of these paths may produce the expected outputs and some may not. Our goal is to find a TA that is both correct, i.e. produces the same outputs as the SUL, and is deterministic. Consequently, we reward these properties with positive fitness.

The simulation function $\textsc{sim}(\mathcal{G}, tt)$ simulates a timed trace tt on a generated TA $\mathcal{G}$ and returns a set of timed traces. It builds the basis for fitness computation and uses the TTS semantics but does not treat outputs as urgent outputs. From the initial state $(l_0, \mathbf{0})$, where l_0 is the initial location of $\mathcal{G}$, it performs the following steps for each $t_i e_i \in tt$ with $t_0 = 0$:

1. From state $q = (l, \nu)$
2. Delay for $d = t_i - t_{i-1}$ to reach $q^d = (l, \nu + d)$
3. If $e_i \in \Sigma_I$, i.e. e_i is an input:
   - 3.1. If $\exists o \in \Sigma_O, d^o \leq d : (l, \nu + d^o) \xrightarrow{o}$, i.e. an output would have been possible while delaying or an output is possible at time t_i

     $\rightarrow$ then mark e_i
   - 3.2. If $\exists q^1, q^2, q^1 \neq q^2 : q^d \xrightarrow{e_i} q^1 \wedge q^d \xrightarrow{e_i} q^2$

     $\rightarrow$ then mark e_i
   - 3.3. For all q' such that $q^d \xrightarrow{e_i} q'$

     $\rightarrow$ carry on exploration with q'
4. If $e_i \in \Sigma_O$, i.e. e_i is an output:
   - 4.1. If $\exists o \in \Sigma_O, d^o < d : (l, \nu + d^o) \xrightarrow{o}$, i.e. an output would have been possible while delaying

     $\rightarrow$ stop exploration
   - 4.2. If $\exists q^1, q^2, q^1 \neq q^2 : q^d \xrightarrow{e_i} q^1 \wedge q^d \xrightarrow{e_i} q^2$ or $\exists o, o \neq e_i : q^d \xrightarrow{o}$

     $\rightarrow$ stop exploration
   - 4.3. If there is a q' such that $q^d \xrightarrow{e_i} q'$

     $\rightarrow$ carry on exploration with q'

The procedure shown above allows for two types of non-determinism. During delays and before executing an input, we may ignore outputs (3.1.) and we may explore multiple paths with inputs (3.3.). We mark these inputs as non-deterministic (3.1. and 3.2.). Since we explore multiple paths, a single input e_i may be marked along one path but not marked along another path. In contrast, we do not explore non-deterministic outputs, leading to lower fitness for respective traces. By ignoring non-deterministic outputs, we avoid issues related to trivial TA which produce every output all the time. These TA would completely simulate all traces non-deterministically, but would not be useful.

During exploration, $\textsc{sim}(\mathcal{G}, tt)$ collects and returns timed traces tts, which are prefixes of tt but with marked and unmarked inputs. For fitness computation, we apply four auxiliary functions. The first one assigns a verdict to simulations and it is defined as follows.

$$\text{Let } tts = \textsc{sim}(\mathcal{G}, tt) \text{ be collected timed traces}$$

$$\textsc{verdict}(tts, tt) = \begin{cases} \texttt{PASS} & \text{if } |tts| = 1 \wedge tt \in tts \\ \texttt{NONDET} & \text{if } |tts| > 1 \wedge \exists tt' \in tts : |tt'| = |tt| \\ \texttt{FAIL} & \text{otherwise} \end{cases}$$

The simulation verdict is `PASS` if $\mathcal{G}$ behaves deterministically and produces the expected outputs. It is `NONDET`, if $\mathcal{G}$ produces the correct outputs along at least one execution path, but behaves non-deterministically. Otherwise it is `FAIL`. A TA, which produces a `PASS` verdict for all timed traces, behaves equivalently to the SUL for these traces.

The function $\textsc{steps}(tts)$ returns the maximum number of unmarked inputs in a trace in tts, i.e. the number of deterministic steps, and $\textsc{out}(tts)$ returns the number of outputs along the longest traces in tts. Finally, $\textsc{size}(\mathcal{G})$ returns the number of edges of $\mathcal{G}$.

Fitness Computation. In order to compute the fitness of $\mathcal{G}$, we assign the weights $w_{\texttt{PASS}}$, $w_{\texttt{NONDET}}$, $w_{\texttt{FAIL}}$, w_{STEPS}, w_{OUT}, and w_{SIZE} to the information gathered about $\mathcal{G}$. Basically, we give some positive fitness for deterministic steps, correctly produced outputs, and verdicts, but we penalise size. Let $\mathcal{TT}$ be the timed traces on which $\mathcal{G}$ is evaluated. The fitness $\text{FIT}(\mathcal{G})$ is then (note that $w_{\text{VERDICT}(tts)}$ evaluates to one of $w_{\texttt{PASS}}$, $w_{\texttt{NONDET}}$, or $w_{\texttt{FAIL}}$):

$$\text{FIT}(\mathcal{G}) = \sum_{tt \in \mathcal{TT}} \text{FIT}(\mathcal{G}, tt) - w_{\text{SIZE}}\ \text{SIZE}(\mathcal{G}) \ \text{ where}$$

$$\text{FIT}(\mathcal{G}, tt) = w_{\text{VERDICT}(tts)} + w_{\text{STEPS}}\ \text{STEPS}(tts) + w_{\text{OUT}}\ \text{OUT}(tts) \text{ and } tts = \ \text{SIM}(\mathcal{G}, tt)$$

Fitness evaluation adds further parameters and we identified guidelines for choosing them adequately. We generally set $w_{\texttt{FAIL}} = 0$ and use w_{OUT} as the basis for other weights. Usually, we set $w_{\text{STEP}} = w_{\text{OUT}}/2$ and $w_{\texttt{PASS}} = k \cdot l \cdot w_{\text{OUT}}$, where l is the average length of test sequences and k is a small natural number, for instance $k = 4$. More important than the exact value of k is setting $w_{\texttt{NONDET}} = w_{\texttt{PASS}}/2$ which gives positive fitness to correctly produced timed traces but with a bias towards deterministic solutions. The weight w_{SIZE} should be chosen low, such that it does not prevent the creation of necessary edges. We usually set it to w_{STEP}. It needs to be non-zero, though. Otherwise an acceptable solution could be a tree-shaped automaton exactly representing $\mathcal{TT}$ without generalisation. As noted at the beginning of this section, we assign larger fitness to solutions that accept a larger portion of the traces deterministically, as our goal is to learn deterministic TA.

As noted above, a TA $\mathcal{T}$ producing only PASS verdicts behaves equivalently to the SUL with respect to $\mathcal{TT}$, hence $\mathcal{T}$ is "approximately trace equivalent" to the SUL. Due to the restriction to deterministic output-urgent systems, trace inclusion and trace equivalence coincide. As a result, a TA producing a FAIL verdict is neither an under- nor an over-approximation.

8.3.4 Creation of New Population

We discussed how new populations are created at the beginning of this section on the basis of Figure 8.3. In the following, we will present details of the involved steps.

Migration. Initially, we distinguished only passed and failed traces, but later introduced a third verdict for non-deterministically passed traces. In the context of migration, we consider non-deterministically passed traces to be failed. Therefore, $\mathcal{T}_{\text{fail}}$ contains all traces for which the fittest TA of the global population produces a verdict other than PASS. The rationale behind this is that we want to improve the global population for traces with both FAIL and NONDET verdicts.

Selection. We use the same selection strategy for mutation and crossover, except that crossover must not select the same parent twice. In particular, we combine *truncation* and *probabilistic tournament* selection: first, we discard the $(n_{\text{pop}} - n_{\text{sel}})$ worst-performing non-migrated TA (truncation), where n_{sel} is the only paramter of truncation selection. This parameter is initially set to a user-defined value.

After truncation, we perform a probabilistic tournament selection for each selection of an individual. The pool for this form of selection includes the remaining n_{sel} TA of the global population and the migrated TA. Probabilistic tournament selection [139] randomly chooses a set of n_{t} TA and sorts them according to their fitness. It then selects the i^{th} TA with probability p_i, which we set to $p_i = p_{\text{t}}(1-p_{\text{t}})^{i-1}$ for $i \in [1\,..\,n_{\text{t}}-1]$ and to $p_{n_{\text{t}}} = (1-p_{\text{t}})^{n_{\text{t}}-1}$, where p_{t} and n_{t} are parameters of the tournament selection. We fix these parameters to $n_{\text{t}} = 10$ and $p_{\text{t}} = 0.5$.

Truncation selection is mainly motivated by the observation that it increases convergence speed during early generations by focusing search on the fittest TA. However, it can be expected to cause a larger loss of diversity than other selection mechanisms [63]. As a result, search may converge to a suboptimal solution, because TA that might need several generations to evolve to an optimal solution are simply discarded through truncation. Therefore, we gradually increase n_{sel} until it becomes as large as n_{pop}

Table 8.2: Mutation operators

Name	Short description
add constraint	add a guard constraint to an edge
change guard	select an edge and create a random guard if the edge does not have a guard, otherwise mutate a constraint of its guard
change target	change the target location of an edge
remove guard	remove either all or a single guard constraint from an edge
change resets	remove clocks from or add clocks to the clock resets of an edge
remove edge	remove a selected edge
add edge	add an edge connecting two randomly chosen existing locations
sink location	add a new location
merge location	merge two locations
split location	split a location l by creating a new location l' and redirecting an incoming edge of l to l'
add location	add a new location and two edges connecting the new location to existing locations
split edge	replace an edge e with either the sequence $e' \cdot e$ or $e \cdot e'$ where e' is a new random edge (adds a location to connect e and e')

such that no truncation is applied in later generations. For the same reason, we do not discard migrated TA, since they may possess valuable features.

Application of Mutation Operators. We implemented mutation operators for changing all aspects of TA, such as adding and removing clock constraints. Table 8.2 lists all implemented mutation operators for TA. Whenever an operator selects an edge or a location, the selection is random with a bias towards locations and edges which are associated with faults and non-deterministic behaviour. We augment TA with such information during fitness evaluation. To create an edge, we create random guards, create reset sets and choose a random label, as in the initial creation of TA.

The mutation operators form three groups separated by bold horizontal lines. The first and largest group contains basic operators, which are sufficient to create all possible automata. The second group containing *merge location* and *split location* is motivated by the basic principles behind automata learning algorithms. Passive algorithms, such as IOALERGIA [199], often start with a tree-shaped representation of traces and transform this representation into an automaton via iterated state-merging [95]. Active learning algorithms, such as L^* [37], on the other hand usually start with a low number of locations and add new locations if necessary. This can be interpreted as splitting of existing locations, as in the TTT algorithm [151]. Note that this intuition also served as a basis for our fault-based test-case generation for active automata learning that we presented in Chapter 4 [17]. The last two operators are motivated by observations during experiments: *add location* increases the automaton size but avoids creating deadlock states, unlike the operator *sink location*. *Split edge* addresses issues related to input enabledness, where an input i is implicitly accepted without changing state, although an edge labelled i should change the state. The operator aims to introduce such edges. For a single mutation, we generally select one of the mutation operators uniformly at random.

Mutation Strength. To control mutation strength, we augment each TA with a probability p_{mut}. Basically, we perform iterated mutation and stop with p_{mut} after each application of a mutation operator. Hence, low values of p_{mut} cause strong mutation.

TA created by mutation are either assigned the parent's p_{mut}, p_{mut} increased by multiplication with $\frac{10}{9}$, or p_{mut} decreased by multiplication with $\frac{9}{10}$. These changes are constrained to not exceed the range $[0.1, 0.9]$ and each of the three choices has the same probability. TA created via crossover are assigned the

Algorithm 8.1 Crossover of locations l^1 and l^2

```
 1: $(l^1, l^2) \leftarrow$ current product location
 2: **for all** $l^1 \xrightarrow{g^1,a,r^1} l^{1\prime}$ **do**
 3:     **if** $l^2 \xrightarrow{g^2,a,r^2} l^{2\prime}$ **then**                    ▷ synchronise on label $a$
 4:         ADD $((l^1, l^2) \xrightarrow{choose(g^1,g^2),a,choose(r^1,r^2)} (l^{1\prime}, l^{2\prime}))$
 5:     **else**
 6:         ADD $((l^1, l^2) \xrightarrow{g^1,a,r^1} (l^{1\prime}, rSel(L^2)))$
 7:     **end if**
 8: **end for**
 9: **for all** $l^2 \xrightarrow{g^2,a,r^2} l^{2\prime}$ s.t. $\nexists g^1, r^1, l^{1\prime} : l^1 \xrightarrow{g^1,a,r^1} l^{1\prime}$ **do**
10:     ADD $((l^1, l^2) \xrightarrow{g^2,a,r^2} (rSel(L^1), l_2'))$
11: **end for**
```

average p_{mut} of both parents. In the first generation, we set p_{mut} of all TA to the user-specified $p_{\mathrm{mut}_{\mathrm{init}}}$. The search is insensitive to this parameter as it quickly finds suitable values for p_{mut} via mutation.

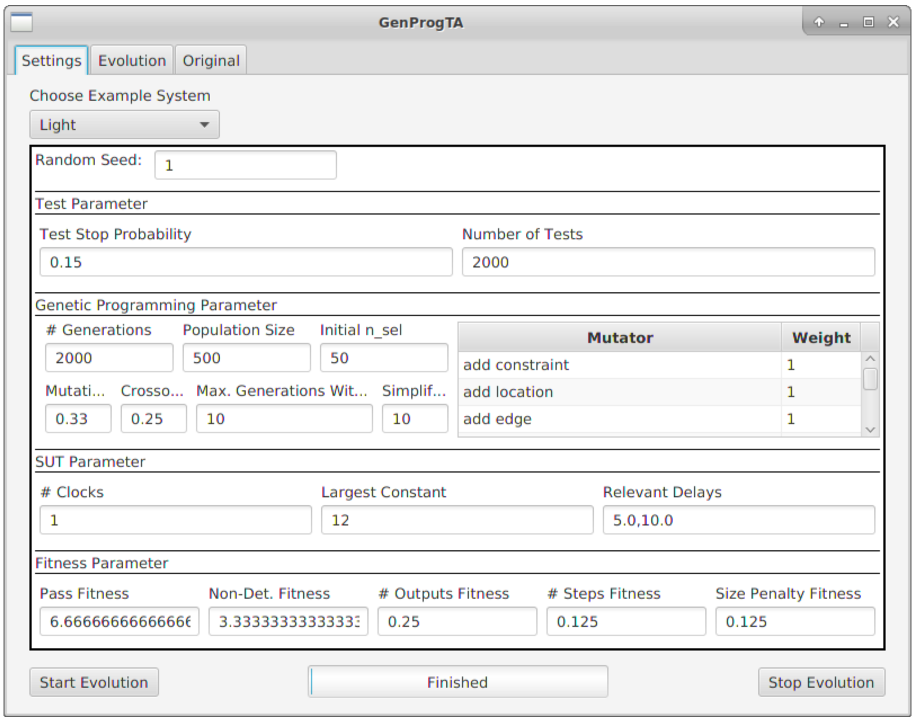
Simplification. In addition to mutation, we apply a simplification procedure. The procedure changes the syntactic representation of TA without affecting semantics, for instance, by removing unreachable locations and self-loop edges for inputs which do not reset clocks. This limits the search to relevant parts of the search space ensuring that unreachable edges are not mutated. The parameter g_{simp} specifies the number of generations between simplifications. Note that we only check the graph underlying the TA, but we do not consider clock values to ensure fast operation.

Crossover. We implemented crossover as a randomised product of two parents. It works as follows. Let L^1 and L^2 be the locations of the two parents and let ${l_0}^1$ and ${l_0}^2$ be their respective initial locations, then the locations of the offspring are given by $L^1 \times L^2$. Beginning from ${l_0}^1$ and ${l_0}^2$ and the initial product location $({l_0}^1, {l_0}^2)$, we explore both parents in a breadth-first manner and we add edges via the algorithm shown in Algorithm 8.1. The algorithm synchronises on action labels and adds edges common to both parents, while randomly choosing the guard and reset set from one of the parents. Edges present in only one parent (Line 6 and Line 10) are added as well, but the target location for the other parent is chosen randomly. As in previous chapters, we apply *rSel* for uniformly random selections. Based on that, the auxiliary function CHOOSE$(a, b) = rSel(\{a, b\})$ returns either a or b with equal probability and $rSel(L)$ chooses a location in L uniformly at random.

To avoid creating excessively large offspring, we stop the exploration and the addition of edges, once the number of reachable product locations is equal to $\max(L^1, L^2)$. As a result, the offspring may not have more locations than both parents. The reachability check only considers the graph underlying the TA and ignores guards due to efficiency reasons.

8.3.5 Implementation

The presented algorithms have been implemented in a tool using Java 8. The tool can be found online [262] and screenshots of it are shown in Figures 8.4 and 8.5. It supports customisation of almost all relevant parameters. When selecting one of the presented experiments, the tool will propose the same values that were used in the evaluation presented in Section 8.4. While the tool is general enough to learn from any set of timed traces given in the correct format, the prototype is currently only meant for evaluating the examples presented in this chapter.



Choose Example System: Light

Random Seed: 1

Test Parameter

Test Stop Probability	Number of Tests
0.15	2000

Genetic Programming Parameter

# Generations	Population Size	Initial n_sel
2000	500	50

Mutati...	Crosso...	Max. Generations Wit...	Simplif...
0.33	0.25	10	10

Mutator	Weight
add constraint	1
add location	1
add edge	1

SUT Parameter

# Clocks	Largest Constant	Relevant Delays
1	12	5.0,10.0

Fitness Parameter

Pass Fitness	Non-Det. Fitness	# Outputs Fitness	# Steps Fitness	Size Penalty Fitness
6.66666666666666	3.33333333333333	0.25	0.125	0.125

Figure 8.4: A screenshot of the parameter-settings tab of the genetic-programming tool for timed automata

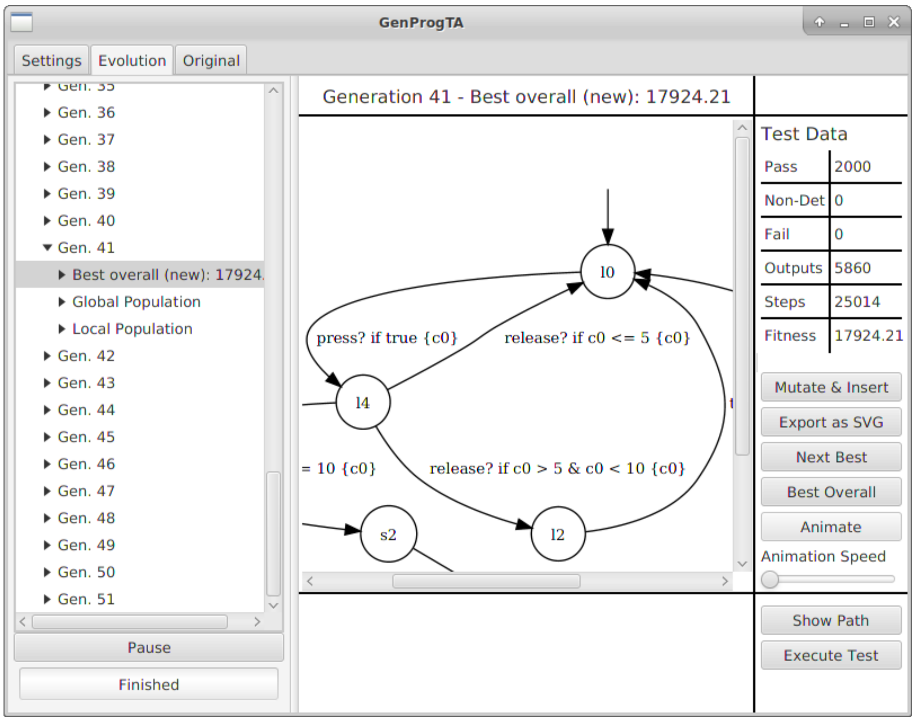
The tool implements the genetic-programming process, with the possibility to inspect the current status of the search, such as the passed timed traces by the current population. In case the search gets stuck, the tool also allows the user to perform manual changes, which enables semi-automatic modelling.

8.4 Case Studies

The genetic-programming evaluation is based on four manually created and 40 randomly generated TA, which serve as our SULs. Using known TA provides us with an easy way of checking whether we found the correct model. It also allows us to simulate time for fast experimentation. However, our approach and our tool are general enough to work on real black-box implementations. As previously mentioned, the approach has been implemented in Java. The implementation includes a demonstrator with a graphical user interface that is available online as part of the supplementary material on learning timed automata [262]. The demonstrator allows users to repeat all experiments presented in the following with freely configurable parameters. The supplementary material additionally includes Graphviz dot-files of the TA used in the evaluation.

For the evaluation, we generated timed traces by simulating n_{test} random test sequences on the SULs. The inputs in the test sequences were selected uniformly at random from the available inputs. The lengths of the test sequences are geometrically distributed with a parameter p_{test}, which is set to 0.15 unless otherwise noted. To avoid creating trivial timed traces, we ensure that all test sequences cause at least one output to be produced. Short test sequences not leading to an output are actually discarded by that, which changes the test-sequence length distribution slightly.

The delays in test sequences were chosen probabilistically in accordance with the user-specified largest constant c_{max}. Additionally, delays close to important constants of the SULs were favoured during test-sequence generation. In practice, one could specify constants gathered from a requirements document if available. Since we simulated known TA for learning, we collected the constants used in guards for that. Specifying appropriate delays generally helps to ensure that the SULs are covered sufficiently well by the test sequences.



Pass	2000
Non-Det	0
Fail	0
Outputs	5860
Steps	25014
Fitness	17924.21

Figure 8.5: A screenshot of the evolution tab of the genetic-programming tool for timed automata

Measurement Setup and Criteria. The measurements were done on a Lenovo Thinkpad T450 with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz. Our main goal is to show that we can learn models in a reasonable amount of time. Improvements of learning efficiency may be possible, for instance, via parallelisation.

We use a training set and a test set for evaluation, each containing n_{test} timed traces. First, we learn from the training set until we find a TA which produces a PASS verdict for all traces. Then, we simulate the traces from the test set and report all traces leading to a verdict other than PASS as erroneous. Note that since we generate the test set traces through testing, there are no negative traces. Similar to the training traces, all traces are observable and can be considered positive. Consequently, notions such as precision and recall do not apply to our setting.

Our four manually created TA, with number of locations and $c_{\max}$ in parentheses, are called car alarm system (CAS) $(14, 30)$, Train $(6, 10)$, Light $(5, 10)$, and particle counter (PC) $(26, 10)$. All of them use one clock. The CAS is an industrial case study, which served as a benchmark for test-case generation for timed systems [20]. A simplified, discrete-time Mealy-machine version of the CAS is shown in Example 2.1 in Section 2.1. Different versions of the Train and Light have been used as examples in real-time verification [49] and variants of them are distributed as demo examples with the real-time model checker UPPAAL [178] and the real-time testing tool UPPAAL TRON [138]. Our version of the Train example is shown in Figure 8.1. The particle counter (PC) is the second industrial case study. Discrete-time versions of it have been examined in model-based testing [21].

In addition to the manually created timed systems, we consider four categories of random TA, each containing ten TA: C15/1, C20/1, C6/2, C10/2, where the first number specifies the number of locations and the second number is the number of clocks. TA from the first two categories have alphabets containing 5 distinct inputs and 5 distinct outputs, while the TA from the other two categories have 4 inputs and 4 outputs. For all random TA, we have $c_{\max} = 15$.

We used similar configurations for all experiments. Following the suggestions in Section 8.3, we set the fitness weights to $w_{\text{OUT}} = 0.25$, $w_{\text{STEPS}} = \frac{w_{\text{OUT}}}{2} = w_{\text{SIZE}}$, $w_{\texttt{PASS}} = \frac{4 w_{\text{OUT}}}{p_{\text{test}}}$, $w_{\texttt{NONDET}} = \frac{\texttt{PASS}}{2}$, and $w_{\texttt{FAIL}} = 0$, with the exception of CAS. Since the search frequently got trapped in local fitness maxima with non-deterministic behaviour, we set $w_{\text{OUT}} = \frac{w_{\text{STEPS}}}{2}$ and $w_{\texttt{NONDET}} = -0.5$. Through these

Table 8.3: Measurement results for learning timed automata via genetic programming

TA	test set errors	generations	time
CAS	0	147/246.0/305.8/595	$27.3m/57.2m/1.2h/2.7h$
Train	0	50/71.0/83.4/180	$2.9m/4.7m/4.8m/9.1m$
Light	0	42/77.5/84.5/240	$3.2m/7.4m/8.7m/31.1m$
PC	0	278/685.5/554.9/859	$3.0h/8.7h/7.3h/10.6h$
C15/1	0/2.0/1.8/6	201/404.5/401.3/746	$1.4h/3.1h/3.2h/6.6h$
C20/1	0/0.0/1.0/6	45/451.0/665.8/1798	$23.4m/6.7h/7.4h/18.3h$
C6/2	0/0.0/0.5/3	18/68.5/176.9/709	$9.4m/43.9m/1.8h/7.6h$
C10/2	0/2.5/2.6/8	73/239.0/344.9/984	$35.8m/3.1h/3.4h/9.3h$

settings, we assign larger fitness to deterministic steps than to outputs and we add a small penalty for non-determinism. Other than that, we set $g_{\max} = 3000$, $n_{\text{pop}} = 2000$, the initial $n_{\text{sel}} = \frac{n_{\text{pop}}}{10}$, $n_{\text{test}} = 2000$, $p_{\text{cr}} = 0.25$, $g_{\text{change}} = 10$, $p_{\text{mutinit}} = 0.33$, and $g_{\text{simp}} = 10$, with the following exceptions. Train and Light require less effort, thus we set $n_{\text{pop}} = 500$. The categories C10/2, C15/1, and C20/1 require more thorough testing, so we configured $n_{\text{test}} = 4000$ for C10/2 and C15/1, and $n_{\text{test}} = 6000$ with $p_{\text{test}} = 0.1$ for C20/1. We determined the settings for n_{test} experimentally, by manually inspecting if the intermediate learned TA were approximately equivalent to the true models, so as to ensure that the training sets adequately cover the relevant behaviour.

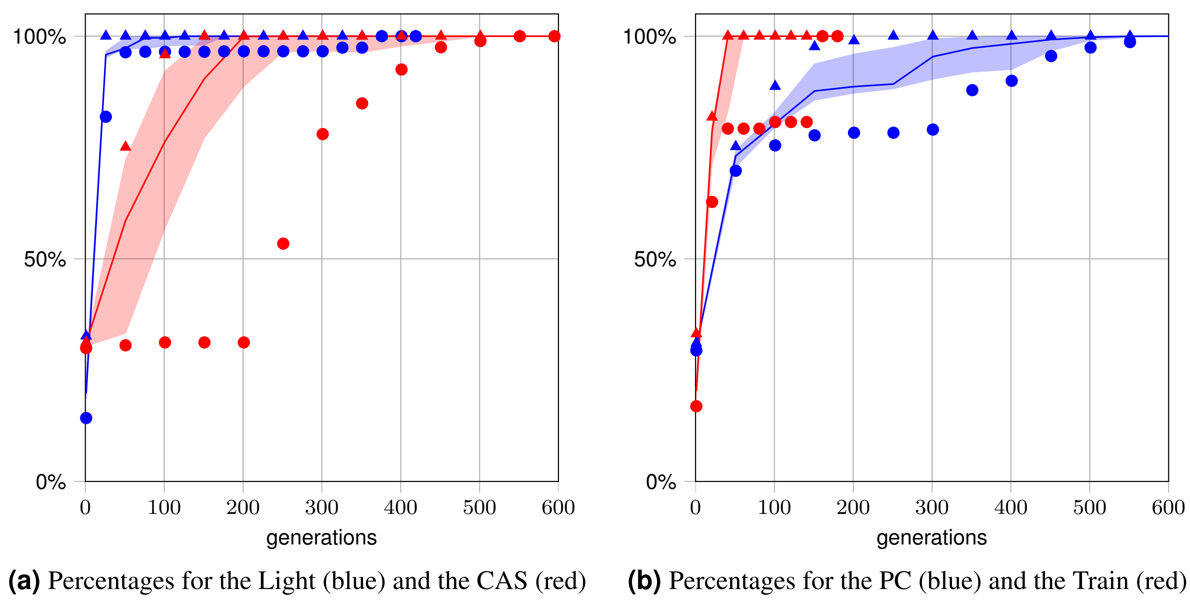
Altogether, we performed 80 learning runs: 10 repeated runs for each manual TA and 10 for each random category, i.e. one run per random TA. All learning runs were successful at finding a TA without errors on the training set, except in two cases, one in category C10/2 and one in category C20/1. In the first case, we repeated the learning run with a larger population $n_{\text{pop}} = 6000$, resulting in successful learning. For the random TA in C20/1, we observed a similar issue as for CAS. Non-determinism was an issue, but we used another solution to counter that. In some cases, crossover may introduce non-determinism, thus we decreased the crossover probability p_{cr} to 0.05 which resulted in successful learning. Hence, we are able to learn TA that are consistent with given trace data via genetic programming.

Results. Table 8.3 shows the learning results. The column *test set error* contains 0, if there were no errors on the test set. Otherwise, each cell in the table contains, from left to right, the minimum, the median and the mean, and the maximum computed over 10 runs for manually created TA and over 10 runs for each random category.

Figure 8.6a and Figure 8.6b illustrate the percentage of correct steps when simulating the timed traces from the training set on the fittest intermediate learned models. The solid line represents the median out of the 10 runs, the dots represent the minimum, the triangles the maximum and the coloured area is the area between first and third quartile. One can see a steep rise in the early generations, while later generations are mainly needed to minimise the learned models that already correctly incorporate all test steps. The CAS is the model with the slowest initial learning, where, in the worst case, the first 200 generations did not improve the model.

The test set errors are generally low, so our approach generalises well and does not simply overfit to the training data. We also see that manually created systems produced no test set errors, while the more complex random TA led to errors. However, for them the relative number of errors was at most two thousandths (8 errors out of 4000 tests). Such errors may, for instance, be caused by slightly too loose or too strict guards on inputs.

The computation time of at most 18.3 hours seems acceptable, especially considering that fitness evaluation, as the most time-consuming part, is parallelisable. Finally, we want to emphasise that we identified parameters which almost consistently produced good results. In the exceptions where this was not the case, it was simple to adapt the configuration.



(a) Percentages for the Light (blue) and the CAS (red) **(b)** Percentages for the PC (blue) and the Train (red)

Figure 8.6: Evolution of the percentages of accepted test steps of the fittest timed automata

The size of our TA in terms of number of locations ranges between 5 and 26. To model real-world systems, it is therefore necessary to apply abstraction during the testing phase, which samples timed traces. Since model learning requires thorough testing, abstraction is commonly used; see Section 2.5. We generally consider similarly-sized systems in this thesis. Consequently, the requirement of abstraction is not a severe limitation.

Manual Interpretability. Figure 8.7 shows a learned model of the CAS as it is produced by our tool without any post-processing. It is observationally equivalent to the true system. Hence, there is no test sequence which distinguishes the SUL from the learned model. Note that the model is also well comprehensible. This is due to the fitness penalty for larger models and due to implicit input-enabledness. Both measures target the generation of small models containing only necessary information. The CAS model only contains a few unnecessary clock resets and an ineffective upper bound in the guard of the edge labelled *soundOff!*, but removing any edge would alter the observable behaviour. Therefore, our approach may enable manual inspection of black-box timed systems, which is substantially more difficult and labour-intensive given only observed timed traces.

8.5 Summary

We presented an approach to learn deterministic TA with urgent outputs, which is an important subclass for testing timed systems [137]. The learned models may reveal flaws during manual inspection and enable verification of black-box systems via model checking. Genetic programming, a metaheuristic search-based technique, serves as the basis framework for learning. In our instantiation of this framework, we parallelised search by evolving two populations simultaneously and developed techniques for mutation, crossover, and for a fine-grained fitness evaluation. Due to the heuristic nature of the proposed method, we cannot provide a convergence proof. Nonetheless, we provide empirical evidence that the method performs well, is capable of coping with state spaces sufficient to model practical systems and generally converges to a solution consistent with given trace data. We evaluated the technique on non-trivial TA with up to 26 locations. We could learn all 44 TA models, where only two random TA needed a small parameter adjustment.

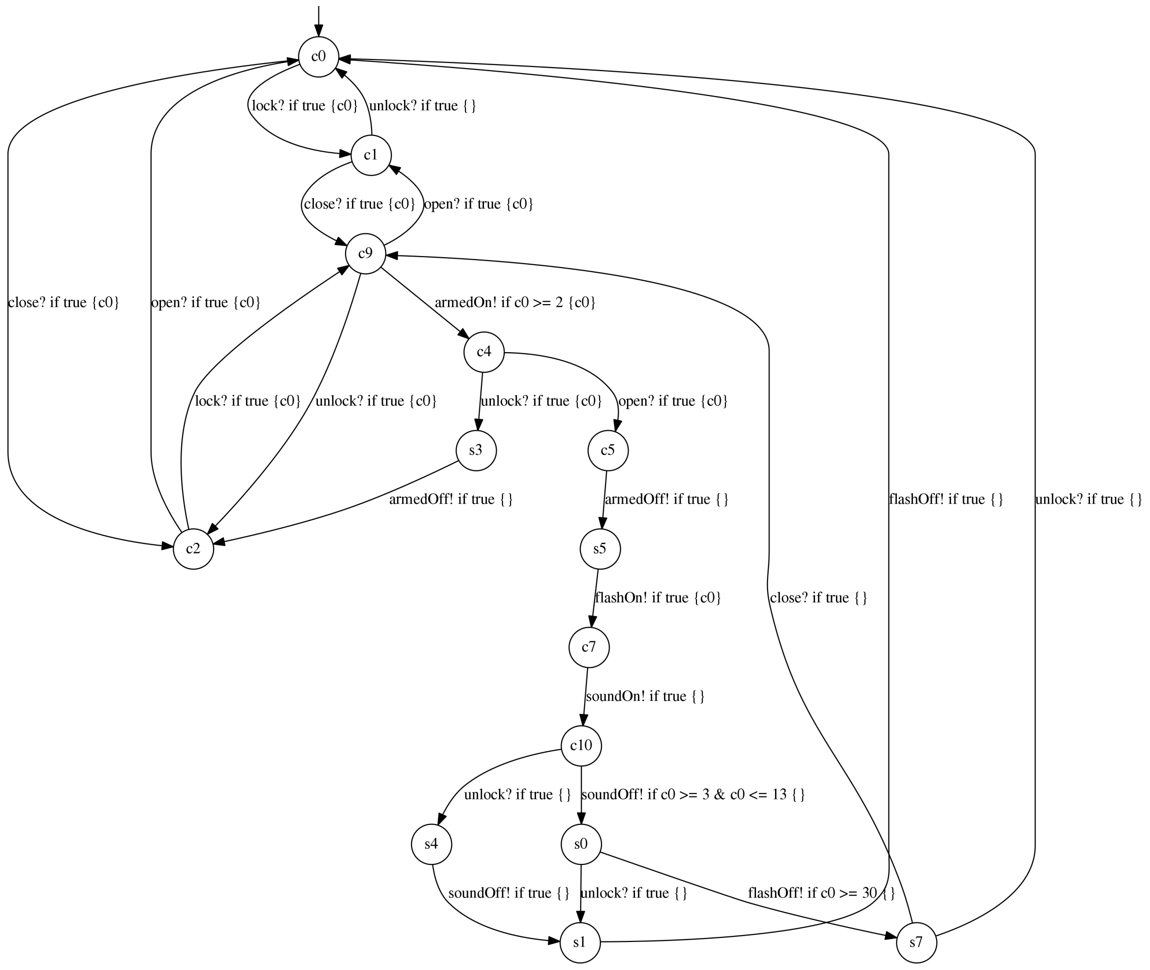


Figure 8.7: A learned model of the car alarm system (CAS)

8.6 Results and Findings

In the following, we review the work presented within this chapter in relation to the research questions defined in Section 1.6.3.

RQ 3.1 Is learning of real-time system models feasible through metaheuristic search-based techniques? Yes, it is possible to learn TA via metaheuristic search-based techniques. We have successfully adapted genetic programming to this modelling formalism and developed a learning technique that reliably converges to a correct model that is consistent with given training data.

Similar to genetic programming in general, our approach has various parameters that affect performance. Therefore, we face the risk that the approach only works under certain configurations for a given type of systems. To counter this risk, we identified parameter settings that work well across all 44 considered SULs. Although we could potentially learn each individual SUL more efficiently, we specifically aimed at a general solution.

RQ 3.2 What assumptions are sufficient to enable learning in the context of real-time systems? It was sufficient to assume determinism, isolated outputs, and output urgency. However, it was not necessary to put assumptions on clock resets. We succeeded at learning TA with more than one clock and arbitrary clock resets. Moreover, we learned input-enabled TA, which makes them well-suited for a testing context.

The question of how to relax our assumptions arises naturally. Determinism and isolated outputs, which is a special form of determinism, are not severe limitations, since determinism is a common assumption in learning of Mealy machines as well. Output urgency, however, does not allow for uncertainty with respect to output timing. Any imprecision that should be allowed in this regard needs to be handled by the component simulating timed traces for the fitness evaluation. If we want to capture uncertain output timing in models, then we need to weaken the assumption of output urgency. As discussed in the context of learnability in Section 8.2.1, our assumptions provide us with negative samples that enable learning.

Hence, a different source of negative samples might enable getting rid of assumptions. This may not be necessary, though. Instead of requiring outputs to be produced as soon as they are enabled, we could require that outputs must be produced within k time units after being enabled. A timed trace containing an output that is produced after $k + 1$ timed units would be a negative example under this assumption.

Concluding Remarks. In conclusion, we have presented a genetic-programming-based technique that reliably learns models of real-time systems which are consistent with given training data. These models generalise to test data that is produced equally, without overlapping with the training data. Since we learn from randomly generated data in our experiments, the learned models may not be equivalent to true underlying models. However, a manual inspection revealed that we generally learned correct models, with the exception of slight discrepancies in behaviour in some cases.

9 Active Genetic Programming of Timed Automata

Declaration of Sources

This chapter presents an overview of an active approach to learning timed automata via genetic programming. For details we refer to Andrea Pferscher's master's thesis [234] on this topic. The master's thesis builds upon the passive learning presented in Chapter 8 and improves learning efficiency with respect to various performance measures through active testing. It was co-supervised by the author of this thesis.

9.1 Introduction

The general theme of this thesis is learning-based testing and test-based learning of black-box systems. While we apply different learning techniques throughout this thesis, we generally consider an active setting. The only exception to that is Chapter 8, which presents a metaheuristic passive learning approach for real-time systems. In this chapter, we apply the passive learning approach from Chapter 8 in an active setting to implement an active learning technique for real-time systems, hence adhering to the theme of this thesis.

While discussing passive learning in Chapter 8, we emphasised the importance of deriving models that accommodate model-based testing. In this chapter, we build upon the passive genetic programming from Chapter 8 to learn approximate hypothesis models. The models serve as the basis for model-based testing. Test observations in turn are used to learn more accurate models. More concretely, we apply the passive genetic programming technique in an active setting by repeatedly interleaving phases of learning and testing. The testing technique developed by Andrea Pferscher [234] performs random walks and roughly follows ideas that we applied in previous chapters; see for instance the test-case generation technique proposed in Chapter 4.

Chapter Structure. The rest of this chapter is structured as follows. We present the main approach and some details on testing in Section 9.2. In Section 9.3, we present selected results from the evaluation of the proposed active learning approach. Section 9.4 provides a summary and Section 9.5 concludes this chapter with a discussion of results and findings.

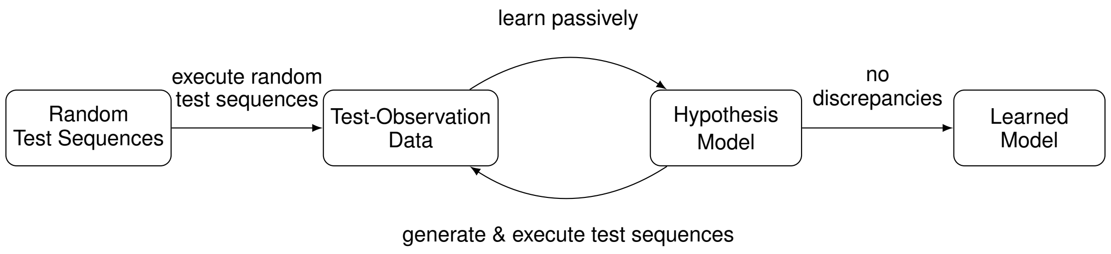


Figure 9.1: Overview of the active genetic programming process

9.2 Method

9.2.1 Moving from Passive Learning to Active Learning

In order to move from passive to active learning, we apply a learning process similar to the one proposed by Walkinshaw et al. [286]. An overview of this process is depicted in Figure 9.1 (see also [234, Figure 3.1]). We start with an initially random set of test sequences which we execute on the SUL. The test-sequence execution produces test observations in the form of timed traces. Given the initial test observations, we invoke the passive genetic programming technique from Chapter 8 to learn a first hypothesis model. From this model, we generate test sequences via random walks and execute these test sequences on the SUL. All timed traces that reveal discrepancies between the hypothesis model and the SUL are added to the test observations. In other words, we extend the test observations by counterexamples to equivalence between hypothesis and SUL. After that, we learn new and more accurate models from the extended test-observation data. This loop of testing and learning is repeated until the test data does not contain any counterexample to equivalence between SUL and hypothesis and we are not able to find new counterexamples via testing. Once we stop, we output the final hypothesis as learned model. This actually resembles active automata learning in the MAT framework without output queries, that is, learning is solely based on equivalence queries.

9.2.2 The Active Genetic Programming Process

In the following, we will take a closer look at the active genetic-programming process. Therefore, we discuss some important details that need to be taken into account when relying on a metaheuristic approach to passive automata learning.

Adaptation of Passive Learning. For the active process, the passive genetic programming has been adapted to not start from scratch with a new population containing individuals with only two locations. Each call of passive learning starts with the final population from the last call, if there is any. The first call starts with a random population containing simple TA as in Chapter 8.

Incomplete Learning. It may happen that passive learning does not find a TA that models the test observations perfectly in reasonable time. In other words, it may take an excessive number of generations to find a model that passes all timed traces stored in the test observations. A potential reason for this kind of issues is low coverage of certain aspects of the SUL's behaviour by the test observations. If the test observations contain little information on some parts of the SUL, the metaheuristic search may not find an appropriate model. For this reason, we limit the number of generations of each iteration of the active process: each call of passive genetic programming may perform up to $g_{\mathrm{max}_{\mathrm{active}}}$ generations.

Hence, if passive learning takes more than $g_{\mathrm{max}_{\mathrm{active}}}$ generations, we use the final timed automaton as next hypothesis, even though it may not pass all sampled timed traces. Subsequent iterations of the active process add further counterexamples to equivalence between SUL and hypothesis. The reasoning for that is the following. The newly added counterexamples potentially also increase the information on parts that are not yet modelled by the hypothesis. Eventually, there will be sufficient information to passively learn an adequate model.

We also need to ensure that we do not add too many traces, as learning performance decreases as the size of the test-observation data grows. Therefore, we add at most n_{fail} traces in each iteration. Finally, we limit the maximum number of generations of the complete active process summed over all iterations by $g_{\max}$. This is necessary as we rely on metaheuristics, for which we do not have a convergence proof. Hence, we stop the process if we either (1) do not find any new counterexamples or (2) we reach the maximum number of iterations $g_{\max}$.

Initial Hypothesis. As discussed by Pferscher [234], there are various options for choosing an initial hypothesis model. As in adaptive model-checking [129], we could use an existing approximate model that may not be up-to-date. In the evaluation, we generally apply the following approach. We create a single random test sequence and execute this sequence to sample a timed trace produced by the SUL. Then, we apply genetic programming to find a timed automaton modelling this trace.

This approach has various benefits. First, it does not require prior knowledge. Second, creating an initial model in this way is very fast. Third, starting from a single trace follows a similar reasoning as the passive genetic programming in general. The passive approach starts the search from very simple timed automata with two locations and it only adds complexity incrementally throughout the search.

9.2.3 Real-Time Test-Case Generation for Active Genetic Programming

Next, we will first discuss the testing process that is performed in each iteration of active genetic programming. Then, we will discuss how individual test cases are generated through random walks.

Testing Process. In each iteration of active genetic programming, we perform Algorithm 9.1 to test the current hypothesis TA $\mathcal{T}$. The algorithm creates up to n_{fail} timed traces $\mathcal{TT}$ witnessing that the SUL and $\mathcal{T}$ are not equivalent.

Algorithm 9.1 executes up to n_{attempts} test sequences to find a single counterexample trace to equivalence, thus it executes at most $n_{\text{fail}} \cdot n_{\text{attempts}}$ test sequences overall. For each test-sequence execution, we first perform a random walk on the hypothesis $\mathcal{T}$ to generate a timed trace *hypTrace* produced by $\mathcal{T}$ (Line 4). Note that for the sake of simplicity we abstract away from details such as further parameters of the function performing the random walk. However, we will comment on important details below in our discussion of random walks. From the trace *hypTrace*, we extract a test sequence containing only inputs, by removing all outputs in Line 5. In Line 6, we execute the extracted test sequence on the SUL which produces another timed trace *sulTrace*.

If *sulTrace* and *hypTrace* are different, then *sulTrace* is a counterexample to equivalence between the SUL and the hypothesis. In this case, we truncate the trace of the SUL to the shortest trace that still is a counterexample to equivalence (Line 8). We add the shortened trace to the set of traces $\mathcal{TT}$ that show non-equivalence. In Line 13, we return $\mathcal{TT}$ which contains all found counterexamples.

In summary, we generate test sequences one by one by performing walks. Then, we execute those test sequences to determine whether they reveal non-equivalence between SUL and hypothesis. If they do, we add them to the set of timed traces that will be used for learning.

Random Walks. On an abstract level, test sequences are generated by random walks on the TTS capturing the semantics of the current hypothesis. We alternate between timed and discrete actions that are possible in the hypothesis, but we favour certain transitions over others. In other words, the probabilistic choices performed during the random walks are biased. We also perform additional checks.

Algorithm 9.1 Testing process for active genetic programming of timed automata

```
Input: $\mathcal{T}$: hypothesis TA, $\mathcal{S}$: SUL
Output: $\mathcal{TT}$: timed traces showing non-equivalence between $\mathcal{T}$ and the SUL $\mathcal{S}$
 1: $\mathcal{TT} \leftarrow \emptyset$
 2: for $i \leftarrow 1$ to $n_{\mathrm{fail}}$ do
 3:     for $i \leftarrow 1$ to $n_{\mathrm{attempts}}$ do
 4:         $hypTrace \leftarrow$ RANDOMWALK($\mathcal{T}$)
 5:         $testSeq \leftarrow$ EXTRACTTESTSEQUENCE($hypTrace$)
 6:         $sulTrace \leftarrow$ EXECUTE($\mathcal{S}, testSeq$)
 7:         if $hypTrace \neq sulTrace$ then
 8:             $minTrace \leftarrow$ shortest $t \in prefixes(sulTrace)$ s.t. $t \neq hypTrace[\leq |t|]$
 9:             $\mathcal{TT} \leftarrow \mathcal{TT} \cup \{minTrace\}$
10:         end if
11:     end for
12: end for
13: return $\mathcal{TT}$
```

The following considerations affect the effectiveness of generated test sequences with respect to their capability of detecting non-equivalence.

Implicit input enabledness: recall that our TA are implicitly input enabled. On the level of semantics, we add self-loop transitions for undefined inputs. Because of that, a hypothesis TA generally accepts all inputs in the test-observation data. As a result, hypotheses may lack input edges that are actually needed. This happens especially for inputs, where the test-observation data does not contain the corresponding outputs produced after those inputs.

For this reason, we also perform input actions that are not enabled (i.e., their guards are not satisfied) in the states visited by the random walk. We call this performing *non-location-changing* actions, as these actions correspond to self loops in the hypothesis. The rationale behind this is that *non-location-changing* actions potentially explore new behaviour. With that, we aim to detect if these actions are truly self loops, or if they lead to an observable change of the SUL state.

The probability of performing this kind of actions is set to be large at the beginning of the active genetic process and decreases in each iteration. Exploring new behaviour is important in the early phase of learning, whereas we assume the hypothesis to be approximately correct in the final phase of learning. In this phase, we aim to test the behaviour specified by the hypotheses more thoroughly.

Choosing delays: during random walks, we apply two strategies to choose delays for timed actions: (1) we choose delays randomly in some prespecified interval, as for the test-sequence generation for the passive approach. Alternatively to that, (2) we also choose delays specified in guard constraints of edges outgoing from locations that are visited by the random walk.

The choice between the two strategies is determined probabilistically during random walks. In the evaluation, we configured the probabilistic choice such that both strategies are equally likely to be applied. Through this strategy it is possible to gradually learn important delays during active genetic programming. Testing can focus on these delays which increases effectiveness.

Zeno behaviour: random walks perform outputs if they are enabled in the currently visited state, because we consider them to occur urgently. By doing that we may enter a loop consisting of output actions, in which time cannot elapse. Behaviour that allows an infinite sequence of discrete actions to be performed without time elapsing is commonly called Zeno behaviour [41]. If we detect this kind of behaviour, we stop the random walk and execute the test sequence corresponding to the generated timed trace on the SUL. In general, this will reveal a discrepancy between the SUL and the hypothesis, because Zeno behaviour is not possible in actual systems. Time needs to elapse at some point.

9.3 Evaluation

In the following, we will discuss selected aspects of the evaluation of active genetic programming of timed automata. The evaluation performed by Pferscher [234] compares the performance of the active genetic programming to the performance of the passive approach on the basis of four evaluation criteria discussed below. Forty random timed automata are considered, as in the evaluation in Section 8.4 and three of the four manually defined timed automata from Section 8.4. Here, we also treat these timed automata as black boxes and simulate them for trace generation. Also as in Section 8.4, we distinguish two sets of timed traces: (1) a training set from which we learn automata and (2) a test set on which we evaluate learned automata.

To investigate the influence of the training set size on the active and the passive approach, we have performed learning experiments for varying training set sizes. In particular, we want to examine if the active approach requires fewer traces than the passive approach to learn correctly. Since learning is affected by random choices, every learning experiment has been repeated ten times and we report statistics computed from that.

9.3.1 Evaluation Setup

Evaluation Criteria. We use the following four different criteria in our evaluation to compare active and passive genetic programming.

Correctness: the percentage of traces in the test set that a learned automaton passes.

Needed training set size: the number of timed traces in the training set. For the active approach, this is the overall number of traces generated by Algorithm 9.1 before stopping. The training set size is fixed to n_{test} for the passive approach and bounded by n_{test} for the active approach.

Test-execution time: the time units that are required for executing the complete training set on the SUL. This is the amount of time specified by the TA, which is simulated in our case.

Learning runtime: the wall-clock time required by learning-related computations, such as fitness evaluation and mutations.

Genetic Programming Configuration. The evaluation configuration for genetic programming is the same as in Section 8.4, except for two differences. The training set size n_{test} ranges between 50 and 2000 with a step size of 100 starting at 100. Furthermore, the crossover probability for active genetic programming is lower. It is set to $p_{\text{cr}} = 0.05$, as this showed better performance in initial experiments. The other parameters of the active approach are set as follows: $g_{\text{max}_{\text{active}}}$ is set to 100, limiting the number of generations in each iteration of active learning to 100, g_{max} is set to 2000, which limits the overall number of generations of active learning. The maximum test-sequence length for the active approach is 40. We have chosen this specific value, as we target systems with up to approximately 30 locations. The probability of choosing a non-location changing action is initially 0.9 and decreased by 0.1 in every iteration of active learning. Finally, we set $n_{\text{attempts}} = 2000$, which is the maximum number of test-sequences that are executed in a single iteration, and we set n_{fail} according to

$$\left\lceil \frac{n_{\text{test}}}{\frac{g_{\text{max}}}{g_{\text{max}_{\text{active}}}} \cdot 0.9} \right\rceil . \tag{9.1}$$

This formula ensures that all timed traces are added during the first 90 per cent of the iterations. The last ten per cent of the iterations do not add new behaviour. Instead, the final iterations focus on learning the behaviour reflected in the test-observation data collected so far. Note that n_{test} varies across experiments.

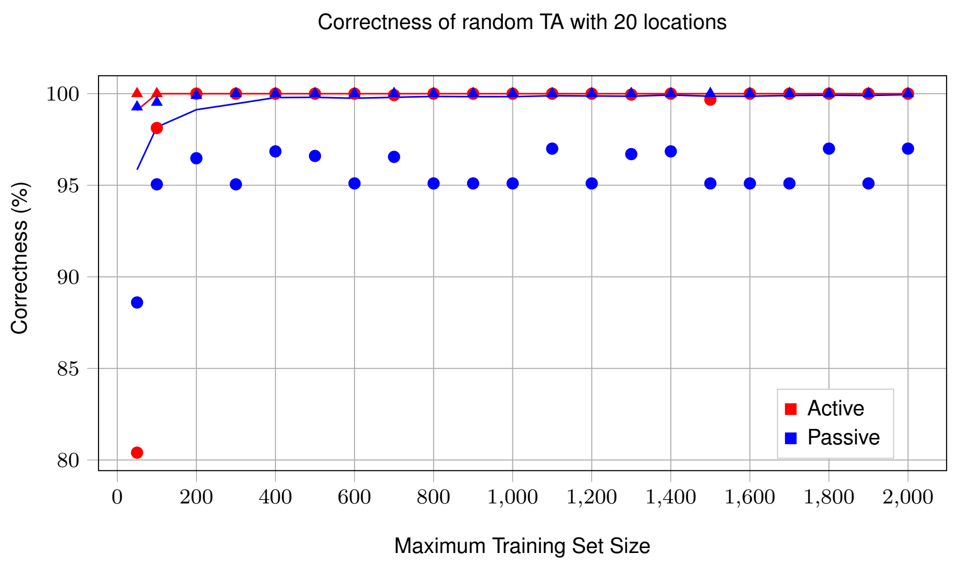


Figure 9.2: Correctness evaluation of the learned TA in the C20/1 category in relation to the training set size

9.3.2 Selected Results

We will present three selected results in the following and we refer to Andrea Pferscher's master's thesis [234] for a thorough discussion of the complete evaluation.

Lower Training Set Size. Figure 9.2 shows plots of measurement results for the correctness criterion for the C20/1 category, with blue denoting results of the active approach and red denoting results of the passive approach. The plots depict statistics derived from the median correctness values computed for the individual random timed automata in the C20/1 category. As this category includes ten random timed automata, the statistics are derived from ten median values at each training set size. Triangles correspond to the maximum median correctness for one approach at a given training set size, where the maximum quantifies over all timed automata in the C20/1 category. Dots correspond to the minimum median correctness and the solid lines denote the median of the median correctness.

We see in Figure 9.2 that the correctness of passive learning approaches 100 per cent, but especially the minimum median correctness is noticeably below 100 per cent. In contrast, active learning requires only at most 200 training traces to achieve complete correctness. Recall that we set the training set size to 6000 for learning the C20/1 category in Section 8.4 and we increased the average training trace length as well. We did this to ensure that the behaviour of the timed automata is sufficiently covered. A lower number of traces may cause some parts to be covered infrequently such that those cannot be adequately modelled by genetic programming. The measurements results shown in Figure 9.2 suggest that this is indeed necessary for passive learning from random timed traces. We need to generate 6000 traces for learning to be feasible. Put differently, passive learning requires 30 times as many training traces as active learning.

Improved Performance. The active genetic programming stops once it cannot find new discrepancies between SUL and hypothesis. Because of that, it may generate a lower number of traces than the passive approach. We shall look at measurement results for the CAS to investigate what that means for learning efficiency. We have chosen the CAS for that, because 100 per cent correctness is achieved with a training set size of 400 by the passive approach and the active approach requires less than 100 traces.

Figure 9.3 and Figure 9.4 show plots of measurement results for the criteria *needed training set size* and *learning runtime*, respectively. Red represents data of the active approach, whereas blue represents

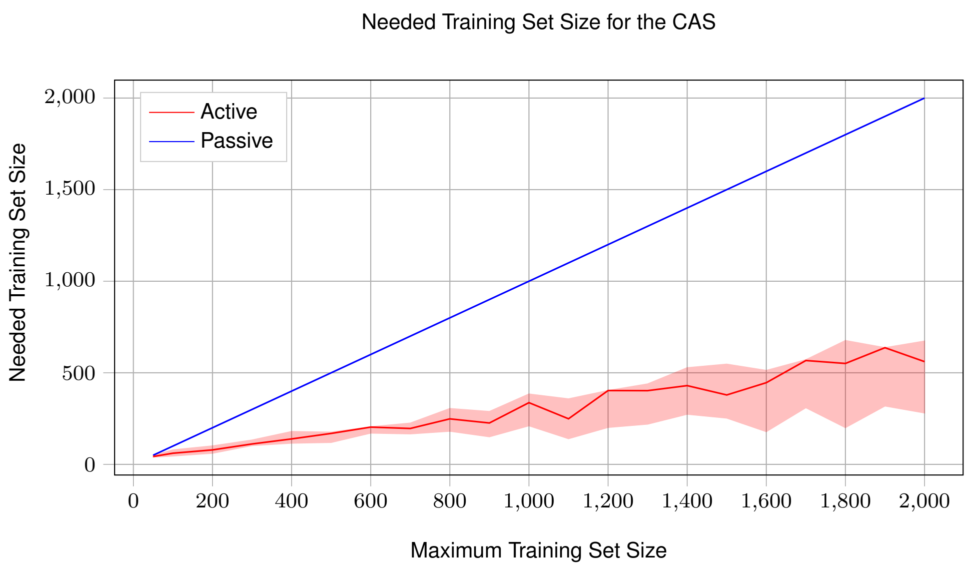


Figure 9.3: Training set size needed for learning a timed automaton model of the CAS

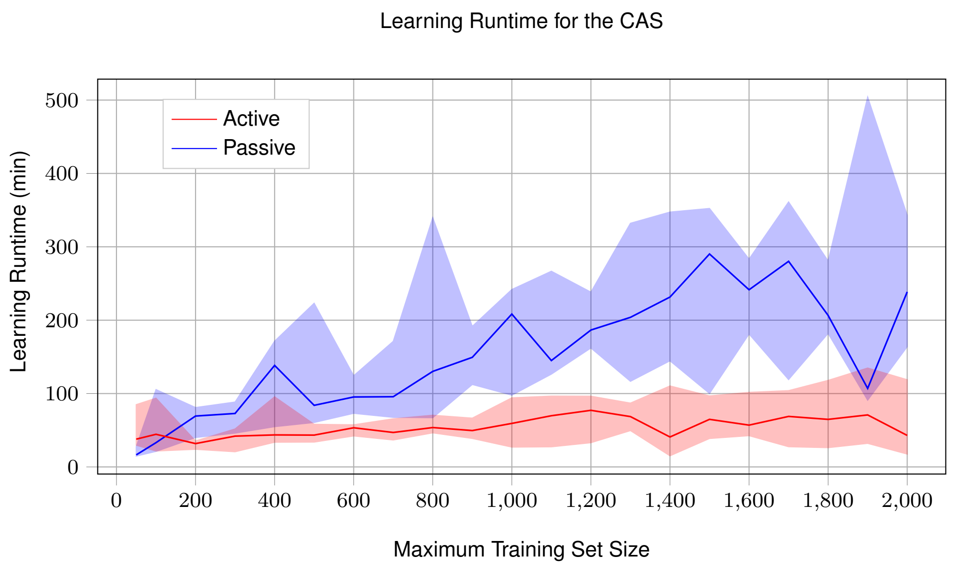


Figure 9.4: Runtime for learning a timed automaton model of the CAS

the passive approach. The solid line shows the median results computed over 10 learning experiments, while the shaded areas represent the interquartile ranges computed from the results, i.e., the range between the first and the third quartile.

We see in Figure 9.3 that active learning generates the maximum number of n_{test} traces when n_{test} is set to 50. For all other values of n_{test} the actual number of generated traces is far below n_{test} which is the fixed training set size for passive learning. Hence, active learning improves upon passive learning also when both learn correctly. However, we also see that the needed training set size slightly grows with n_{test}. This can be explained by considering how n_{fail} is calculated in Equation (9.1). The value of n_{fail} is proportional to n_{test}. Furthermore, note that during early iterations of the active process, it is likely that Algorithm 9.1 finds many counterexamples. Hence, if n_{fail} is large due to large n_{test}, we consequently generate a large number of counterexamples in the early phase of active learning. It may be possible to decrease the needed training set size even further by slightly adapting Equation (9.1). However, computing n_{fail} via Equation (9.1) worked well across all experiments. The measurement results for test-execution time follow a very similar trend as the needed training set size, therefore we do not show them.

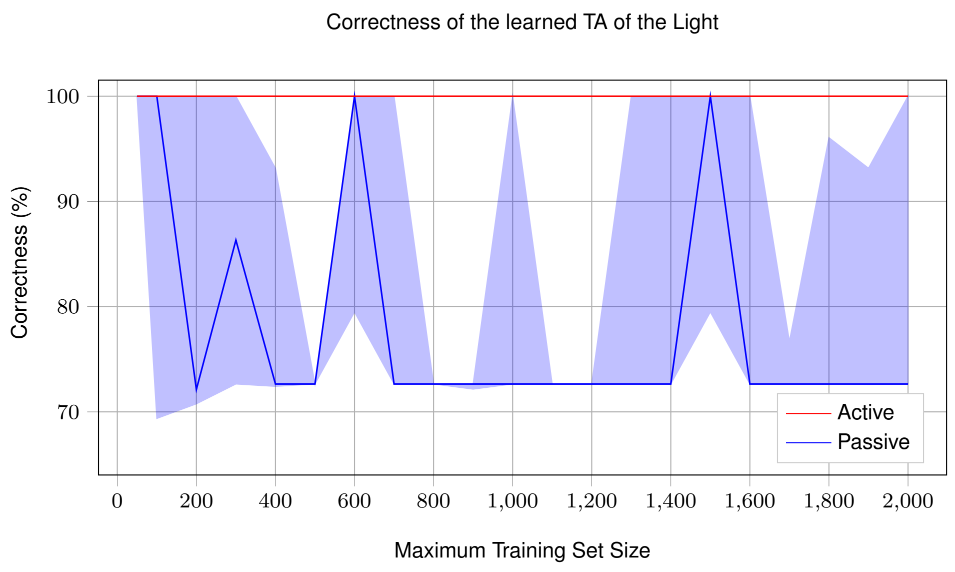


Figure 9.5: Correctness evaluation of the learned TA of the Light example in relation to the training set size

Since active learning generates smaller training sets, fitness computation is significantly faster than for passive learning. As a result, the overall active learning runtime is lower than the runtime of passive learning. This is exactly what we see in Figure 9.4. It is also interesting to observe that the interquartile range of the active learning runtime is much smaller than for passive learning. Some runs of passive learning take much longer than other runs. Active learning can be considered more reliable in this regard.

Local Optima. The evaluation of active genetic programming revealed an issue of passive genetic programming that we did not discover in our original evaluation presented in Section 8.4. Figure 9.5 shows plots of the correctness measurement results for the Light example. The plots again depict the median and the interquartile range. We can observe that the passive approach achieves 100 per cent median correctness only for very few values of n_{test}. In the cases, where passive genetic programming did not achieve 100 per cent correctness, it actually did not learn a model consistent with the given training data. Hence, learning was unsuccessful in many cases. That happened in this evaluation, because the maximum constant $c_{\max}$ was set to a value twice as large as the value used in the evaluation presented in Section 8.4. First, this increases the search space for genetic programming. Second, the delays in timed traces are considerably larger and cover relevant values less frequently. As a result, certain parts of the Light's behaviour seem to be hard to model for the passive genetic programming. We observed that passive learning got stuck in local fitness maxima in many experiments because of that. In these cases, passive learning found timed automata that modelled most of the training set, but any small number of mutations could not increase the fitness. In contrast to this, active learning did not have issues related to local optima, showing another advantage of the active approach.

9.4 Summary

We presented an active learning technique for timed automata in this chapter. The passive genetic programming approach for timed automata introduced in Chapter 8 serves as basis for this active learning technique. The technique iteratively learns models from test observations and it tests learned models in alternation. This process actively learns a timed automaton model and is stopped once it cannot detect new behaviour in the SUL or it reaches a bound on the learning time budget. The test-case generation applied in active learning is implemented through random walks of hypothesis models. This random-walk-based testing strategy is specifically tailored towards learning of timed systems. An evaluation comparing passive genetic programming and active genetic programming demonstrated the favourable performance of the active approach.

The work presented within this chapter is based on Andrea Pferscher's master's thesis [234] which was co-supervised by the author of this thesis. This chapter focused on selected details, challenges, and findings discovered in the evaluation. For details, we refer to Andrea Pferscher's master's thesis on active genetic programming of timed automata [234].

9.5 Results and Findings

In the following, we will discuss the results and insights gained from experiments with active genetic programming of timed automata. As in previous chapters, we will discuss our findings by addressing relevant research questions.

RQ 1.3 Can learning with randomised conformance testing reliably generate correct system models? The evaluation presented in Section 9.3 has demonstrated that we are able to improve the reliability of learning through random-walk-based active testing. Figure 9.2, for instance, shows that we are able to reliably learn accurate timed automata models from relatively few timed traces. In contrast to that, the passive genetic programming approach required approximately 30 times as many traces to learn reliably. Under a limited testing budget for trace generation, the active approach is significantly more reliable than the passive approach.

Given these observations, we conclude that randomised testing is a sensible choice to implement the presented form of active learning, that is, we can answer **RQ 1.1** affirmatively as well.

RQ 3.2 What assumptions are sufficient to enable learning in the context of real-time systems? In both active and passive genetic programming, we assume that the SUL's behaviour can be modelled with deterministic, output-urgent, input-enabled timed automata with isolated outputs. One experiment discussed in Section 9.3 revealed an issue of the passive genetic algorithm with respect to local optima. In the corresponding experiment, we changed the setting of $c_{\max}$, the maximum constant occurring in guards, compared to Section 8.4. Setting it to a very large value increased the search space and caused low coverage of the SUL's behaviour by the training data. We can deduce from the experimental results that for passive learning to converge reliably, we require an approximately correct setting of $c_{\max}$. This *additional assumption* is not needed for active genetic programming. The active technique did not suffer from similar issues.

Concluding Remarks. By applying model-based testing, we transformed the passive genetic-programming-based learning technique presented in Chapter 8 into an active learning technique. The active technique showed favourable performance with respect to several evaluation criteria. For test-case generation in active learning, we follow a similar approach as in the discrete-time setting. For instance, Algorithm 4.1 presented in Chapter 4 for testing Mealy machines also performs random actions and traverses hypothesis models. We applied comparable strategies to test stochastic systems as well. Hence, learning-based testing combining coverage of hypothesis models with randomisation is an effective testing approach for various types of systems.

10 Test-Based Learning of Hybrid Systems

Declaration of Sources

This chapter covers our work on learning-based testing to generate training data for machine learning of hybrid system models. It is based on our conference paper which was selected the best paper at the ICTSS 2019 [29].

10.1 Introduction

This thesis addresses efficient automata learning in networked environments with a special focus on communication protocols. Due to the increasing integration of software-based systems into appliances and devices used in our everyday lives, it is important to study the verification of so-called cyber-physical systems (CPSs) [31, 38, 48, 85, 100, 176, 185, 211, 268]. In CPSs, embedded computers and networks control physical processes. Mostly, CPSs interact with their surroundings based on the context and the (history of) external events through an analogue interface. We also use the term hybrid system to refer to such reactive systems that intermix discrete and continuous components [196]. Hybrid systems are used in consumer electronics as well as in safety-critical areas, such as driving assistance systems used in cars. This highlights the importance of safety assurance techniques for hybrid systems. The manual construction of models that comprise physical and digital behaviour is challenging and often requires expertise in several areas, including control engineering, software engineering and sensor networks [100]. Therefore, the automatic construction of accurate hybrid system models through learning may greatly aid the verification of CPSs.

Hybrid systems are often modelled with hybrid automata [33, 135]. Basically, hybrid automata generalise timed automata, which we learned in the previous two chapters. A hybrid automaton has a finite set of locations (often called modes), discrete events and a set of variables. Similar to the behaviour of timed automata, the behaviour of hybrid automata can be constrained based on variable valuations. While clock valuations in timed automata only increase linearly over time, hybrid automata include flow conditions that govern the evolution of variable evaluations. This increases the complexity of learning considerably. In order to learn general timed automata, it is necessary to find appropriate clock resets, while for hybrid automata it is also necessary to find flow conditions that conform to the hybrid SUL's behaviour.

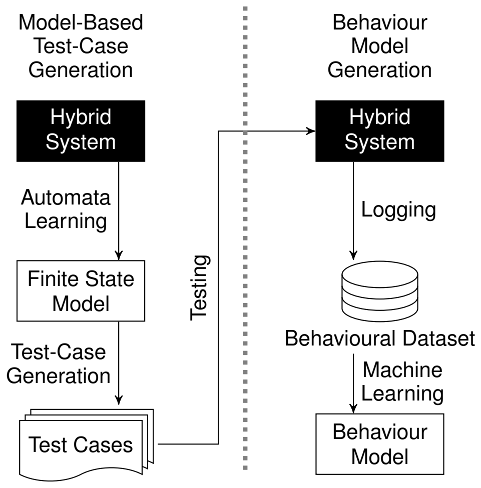


Figure 10.1: Learning a behaviour model of a black-box hybrid system

Learning approaches for hybrid systems from the literature commonly incorporate different phases for the discrete system behaviour and the continuous system behaviour, respectively. Medhat et al. [203] present an L^*-based approach for learning hybrid automata with inputs and outputs. However, it is passive, as it basically treats a training dataset as SUL. The authors assume to be given preprocessed traces of variable valuations that include information of change points corresponding to discrete events. First, they learn a Mealy machine from these traces, then they infer flow conditions and timing relationships. Niggemann et al. [219] propose HYBUTLA, a passive learning algorithm for hybrid timed automata. Like real-time automata learned by Verwer et al. [279], these automata do not distinguish between inputs and outputs and they have a single clock that is reset on every transition. They follow a state-merging-based approach that learns a discrete automaton based on discrete events. This discrete automaton is subsequently extended by functions that model continuous changes of variable values while sojourning in locations.

In this chapter, we propose a two-phase process for learning models of hybrid systems. First, we learn a discrete automaton model of the hybrid system. Then, we learn a recurrent neural network (RNN) model focused on the reachability of specific output events. This RNN model captures both the discrete and the continuous dynamics of the system. Our goal is to learn a RNN-based classifier that can detect whether given input stimuli cause the specified output events to be produced.

In contrast to existing work [203, 219], we do not enhance the learned discrete automaton with information about the non-discrete system dynamics. We rather use the learned automaton to abstractly represent the state space of the system. This abstract representation allows us to explore the state space thoroughly through model-based testing techniques. Test data collected while testing provides us with a large and representative dataset of the behaviour of the SUL. We then use this dataset to train the RNN model, which ensures accurate learning. Through the application of directed testing, we are also able to adequately cover rare side-conditions. This helps to consider safety-critical features during learning. Given the large state space of hybrid systems and the nature of this kind of features, safety-critical features are rarely triggered in random or normal operation. Hence, nominal or random data would be insufficient to accurately learn safety-critical behaviour. For testing we apply advanced test-case generation methods inspired by the testing techniques presented in Chapter 4 (coverage in general) and in Chapter 6 (reachability-directed testing).

Figure 10.1 depicts the execution flow of the complete learning process. In this process, we combine automata learning and model-based testing (MBT) to derive an adequate training set. Then, we use machine learning to learn a behaviour model of the black-box hybrid SUL. We can use the learned behaviour model for various purposes such as monitoring runtime behaviour. Furthermore, it could be used as a surrogate of a complex and heavy-weight simulation model to efficiently analyse safety-critical behaviour offline [252].

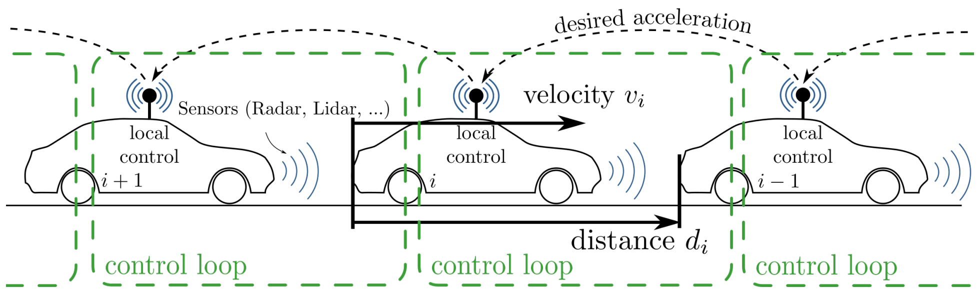


Figure 10.2: Platooning as a distributed control scenario – adapted from a figure by Dolk et al. [101]

Given a black-box hybrid system, we learn automata as discrete abstractions of the system. More concretely, we actively learned Mealy machines from discretised traces collected by testing. Next, we investigate the learned discrete automata for critical behaviours. Once behaviours of interest are discovered, we drive the hybrid system towards these behaviours via model-based testing. During this phase of testing, we record the SUL's output response in a continuous domain, i.e., we do not discretise it. This step in the process creates a behavioural dataset with high coverage of the hybrid system's behaviour including specified rare conditions that should be covered. Finally, we use a machine learning method to train an RNN model that generalises the behavioural dataset. For evaluation, we compare four different test-based approaches to generate data. With each of these approaches, we first generate data via testing. Then, we learn RNN models from the data and compute various performance measures for detecting critical behaviours in unforeseen situations. Experimental results show that RNNs learned with data generated via model-based testing achieved significantly better performance compared to models learned from randomly generated data. In particular, the classification error is reduced by a factor of five and a similar F1-score is accomplished with up to three orders of magnitude fewer training samples.

Example 10.1 (Platooning Overview). Throughout this chapter we illustrate our approach on the basis of a complex networked system. We apply the learning process described above to learn a model of a platooning system focusing on collision detection. In particular, our goal is to learn a behaviour model that can reliably detect collisions. The platooning system is implemented in a testbed in the Automated Driving Lab at Graz University of Technology[1].

Platooning of vehicles is a complex distributed control scenario, where vehicles automatically form platoons; see Figure 10.2. Vehicles in platoons autonomously keep a low distance to their respective successors in order to reduce fuel consumption because of reduced air resistance. Local control algorithms of each vehicle are responsible for reliable velocity and distance control. The vehicles continuously sense their environments. They, for instance, measure the distance to the vehicle ahead and may use discrete, i.e. event triggered, communication to communicate desired accelerations along the platoon [101]. Thus, platooning is implemented by a network of controllers, sensors, and actuators that communicate with their physical environment. As a first step, we consider two vehicles of a platoon, the leader and its first follower, to illustrate our approach. An extension to more vehicles is possible through slight adaptations.

Controller designs are often rigorously analysed on the basis of mathematical models. Mathematical proofs serve to show that important aspects such as stability are satisfied to guarantee safe operation. Given that, it may seem superfluous to learn another model and to test a system that is proven safe. However, learning and testing serve an important role. First, mathematical models are often subject to simplifying assumptions to make proofs tractable. We aim to learn an accurate representation of the implementation of controllers, while placing as few assumptions on the SUL as possible. Testing helps to determine whether the assumptions hold in practice. Second, robustness testing, that is, testing in scenarios violating assumptions, may provide insights into the system behaviour in exceptional situations.

[1]See also `https://www.tugraz.at/institute/irt/research/automated-driving-lab/`, accessed on November 4, 2019

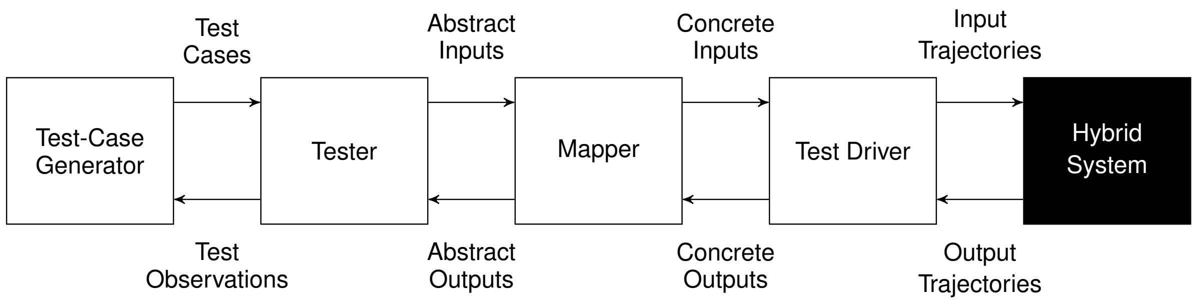


Figure 10.3: Components involved in the testing process

Chapter Structure. The rest of this chapter is structured as follows. Section 10.2 presents the proposed approach. In particular, the section explains the test-case execution, test-case generation, and automata learning for data generation. Additionally, it also briefly explains the RNN training setup. In Section 10.3, we present the results of our evaluation, comparing four different testing techniques to generate behavioural data for the platooning scenario. In Section 10.4, we provide a summary followed by a discussion on our results and findings in Section 10.5.

10.2 Methodology

Our goal is to learn a behaviour model capturing targeted behaviour of a hybrid SUL. The model's response to a trajectory of input variables shall conform to the SUL's response with high accuracy and precision. As in discrete systems, purely random generation of input trajectories is unlikely to exercise the SUL's state space adequately. Consequently, models learned from random traces cannot accurately capture the SUL's behaviour. Therefore, we propose to apply automata learning followed by model-based testing to collect system traces while using a machine learning method for non-discrete, i.e. continuous and hybrid, model learning. In particular, we use RNNs as non-discrete models. Figure 10.1 shows a generalised version of our approach.

The proposed trace-generation approach does not require any prior knowledge, similar to random sampling-based trace generation, whereas it may benefit from domain knowledge and specified requirements. For instance, we do not explore states any further, which already violate safety requirements. In the following, we will first discuss the testing process. This discussion includes the interaction with the SUL, abstraction, automata learning, and test-case generation. Subsequently, we will discuss learning a behaviour model in the form of an RNN with training data collected by executing test cases.

Example 10.2 (Learning Workflow for Platooning). We learn a behaviour model for our platooning scenario in three steps: (1) automata learning which explores a discretised platooning control system to capture the state space structure in learned models, followed by (2) model-based testing exploring the state space of the learned model directed towards targeted behaviour while collecting non-discrete system traces. In step (3), we generalise from those traces by learning an RNN.

10.2.1 Testing Process

We apply various test-case generation methods, with the same underlying abstraction and execution framework. Figure 10.3 depicts the components implementing the testing process.

- **Test-Case Generator.** The test-case generator creates abstract test cases. These test-cases are generated offline as sequences of abstract inputs.
- **Tester.** The tester takes an input sequence and passes it to the mapper. Feedback from test-case execution is forwarded to the test-case generator.

- **Mapper.** The mapper maps each abstract input to a concrete input variable valuation and a duration, defining how long the input should be applied. Concrete output variable valuations observed during testing are mapped to abstract outputs. Each test sequence produces an abstract output sequence which is returned to the tester.

- **Test Driver & Hybrid System.** The test driver interacts with the hybrid system by setting input variable values and sampling output variable values.

System Interface and Sampling

We assume a system interface comprising two sets of real-valued variables: input variables U and observable output variables Y, with U further partitioned into controllable variables U_C and uncontrollable observable input variables U_E affected by the environment. We denote all observable variables by $\mathit{Obs} = Y \cup U_E$. Additionally, we assume the ability to *reset* the SUL, as all test-case execution for trace generation need to start from a unique initial state. During testing, we change the valuations of controllable variables U_C and observe the evolution of variable valuations at fixed sampling intervals of length t_s.

Example 10.3 (Interfacing with the Platooning System). The platooning SUL has been implemented in MathWorks Simulink®. This implementation actually models a platoon of remote-controlled trucks used in the testbed at the Automated Driving Lab, therefore the acceleration values and distance have been downsized. The SUL interface comprises:

- $U_C = \{acc\}$
- $Y = \{d, v_l, v_f\}$
- $U_E = \{\Delta\}$

The leader acceleration '*acc*' is the single controllable input with values ranging from $\frac{-1.5m}{s^2}$ to $\frac{1.5m}{s^2}$. The distance between the leader and the first follower is '*d*'. The velocities of the leader and the follower are 'v_l' and 'v_f', respectively; finally 'Δ' denotes the angle between the leader and the x-axis in a fixed coordinate system given in radians, i.e., it represents the orientation of the leader that changes while driving along the road. Hence, the orientation is an uncontrollable input that depends on the track used for learning.

We sampled values of these variables at fixed discrete time steps, which are $t_s = 250$ milliseconds apart. Note that this sampling rate only affects our interaction with the system, that is, it affects how frequently we change controllable inputs and check observable variable values. The internal controllers and sensors of the platooning system use different sampling rates.

Abstraction

We discretise variable valuations for testing via a mapper. With that, we effectively abstract the hybrid system such that a Mealy machine over an abstract alphabet can model it. Each abstract input is mapped to a concrete valuation of U_C and a duration specifying how long the valuation shall be applied, thus U_C only takes values from a finite set. As abstract inputs are mapped to uniquely defined concrete inputs, this form of abstraction does not introduce non-determinism. In contrast, values of observable variables *Obs* are not restricted to a finite set. Therefore, we group concrete valuations of *Obs* and assign an abstract output label to each group.

The mapper also defines a set called *Violations*. This is a set containing abstract outputs that signal violations of assumptions or safety requirements. In the abstraction to a Mealy machine, these outputs lead to sink states from which the model does not transit away. Such a policy prunes the abstract state space.

A mapper has five components: (1) an abstract input alphabet I, (2) a corresponding concretisation function γ, (3) an abstract output alphabet O, (4) an abstraction function α mapping concrete output values to O, and (5) the set *Violations*. During testing, the mapper performs the following two actions:

- **Input Concretisation.** The mapper maps an abstract symbol $i \in I$ to a pair $\gamma(i) = (\nu, d)$, where ν is a valuation of U_C and $d \in \mathbb{N}$ defines time steps, for how long U_C shall be set according to ν. This pair is passed to the test driver.
- **Output Abstraction.** The mapper receives concrete valuations ν of *Obs* from the test driver and maps them to an abstract output symbol $o = \alpha(\nu)$ in O that is passed to the tester. If $o \in$ *Violations*, then the mapper stores o in its state and maps all subsequent concrete outputs to o until it is reset.

The mapper state needs to be reset before every test-case execution. Repeating the same symbol $o \in$ *Violations* after seeing it once creates sink states, which prunes the abstract state space. Furthermore, the implementation of the mapper contains a cache to return abstract output sequences without SUL interaction if possible. Given an abstract test case $t \in I^*$, cache access is possible in two situations: (1) if we executed a t' before such that t is a prefix of t', or (2) if we executed a t' before such that t' is a prefix of t and t' produced an $o \in$ *Violations*. Cache access in Case (1) is justified by determinism, while cache access in Case (2) is justified by the creation of sink states for each output in the set *Violations*.

Example 10.4 (Abstracting the Platooning Scenario). We tested the SUL with an alphabet I of six abstract inputs: *fast-acc*, *slow-acc*, *const*, *const$_l$*, *brake* and *hard-brake*, concretised by $\gamma(\textit{fast-acc}) = (\textit{acc} \mapsto 1.5m/s^2, 2)$, $\gamma(\textit{slow-acc}) = (\textit{acc} \mapsto 0.7m/s^2, 2)$, $\gamma(\textit{const}) = (\textit{acc} \mapsto 0m/s^2, 2)$, $\gamma(\textit{const}_l) = (\textit{acc} \mapsto 0m/s^2, 8)$, $\gamma(\textit{brake}) = (\textit{acc} \mapsto -0.7m/s^2, 2)$, and $\gamma(\textit{hard-brake}) = (\textit{acc} \mapsto -1.5m/s^2, 2)$. Thus, each input takes two time steps, except for *const$_l$*, which represents prolonged driving at constant speed.

The output abstraction depends on the distance d and the leader velocity v_l. If v_l is negative, we map to the abstract output *reverse*. Otherwise, we partition d into 7 ranges with one abstract output per range. The range $(-\infty, 0.43m)$ is, for instance, mapped to *crash* ($0.43m$ is the length of a remote-controlled truck). We assume that platoons do not drive in reverse. Therefore, we include *reverse* in *Violations*, such that once we observe *reverse*, we ignore the subsequent behaviour. The set *Violations* includes *crash* as well, since we are only interested in the behaviour leading to a crash.

Test-Case Execution

The concrete test-case execution is implemented by a test driver. It basically generates step-function-shaped input signals for input variables and samples output variable values. For each concrete input (ν_j, d_j) applied at time t_j (starting at $t_1 = 0\textit{ms}$), the test driver sets U_C according to ν_j for $d_j \cdot t_s$ milliseconds and samples the values ν'_j of observable variables $Y \cup U_E$ at time $t_j + d_j \cdot t_s - \frac{t_s}{2}$. After that, it proceeds to time $t_{j+1} = t_j + d_j \cdot t_s$ to perform the next input if there is any. In that way, the test driver creates a sequence of sampled output variable values ν'_j, one for each concrete input. This sequence is passed to the mapper for output abstraction.

Viewing Hybrid Systems as Mealy Machines

Test-case execution samples exactly one output value for each input, $\frac{t_s}{2}$ milliseconds before the next input is performed. This ensures that there is an output for each input, such that input sequences and output sequences have the same length. Given an abstract input sequence π_i, the test-case execution produces an output sequence π_o of the same length. In slight abuse of notation, we denote this relationship by $\lambda_h(\pi_i) = \pi_o$. Hence, we view the hybrid system under test on an abstract level as a Mealy machine $\mathcal{H}_m$ with $obs_{\mathcal{H}_m} = \{\langle \pi_i, \lambda_h(\pi_i)\rangle \mid \pi_i \in I^*\}$.

Learning Automata

We applied the active automata learning algorithm by Kearns and Vazirani (KV) [160], implemented by LearnLib [152], in combination with the *transition-coverage* testing strategy described in Chapter 4 [17]. We have chosen the KV algorithm, as it requires fewer output queries to generate a new hypothesis model than L^* [37], such that more equivalence queries are performed. As a result, we can guide testing during equivalence queries more often. We recap the transition-coverage testing strategy briefly below. Although TTT performed better than KV in Section 4.5, KV showed favourable performance in the platooning case study. Using the same number of tests, KV learned larger models.

There are two notable differences between the application of automata learning in this chapter and automata learning in Section 4.5. First, we learn a discrete approximation of a hybrid system. Second, our goal in this chapter is not learning a complete and perfectly correct automaton model. Automata learning rather helps to systematically explore the hybrid SUL's state space. The learned hypothesis models basically keep track of what has already been tested.

Recall that active automata learning operates in rounds, alternating between series of output queries and equivalence queries. We stop this process once we performed the maximum number of tests N_{autl}, where the subscript autl stands for automata learning. The performed tests include both output queries and test queries implementing equivalence queries. Due to the large state space of the platooning SUL, it was infeasible to learn a complete model, hence we stopped learning once we reached the bound N_{autl}, even though further tests could have revealed discrepancies between the SUL and the hypotheses.

Example 10.5 (Learning Automata of the Platooning System). The learned automata provided insights into the behaviour of the platooning SUL. A manual analysis revealed that collisions are more likely to occur, if trucks drive at constant speed for several time steps. Since we aimed at testing and analysing the SUL with respect to dangerous situations, we created the additional abstract input *const*$_l$, which initially was not part of the set of abstract inputs.

During active automata learning, we executed approximately $N_{\mathrm{autl}} = 260000$ concrete test cases on the platooning SUL in 841 learning rounds. These test cases produced 2841 collisions. The fact that we found collisions is actually noteworthy, as the used platooning configuration was assumed to be safe. Hence, this demonstrates that learning-based testing can detect safety violations of complex hybrid systems. In the last learning round, we generated a hypothesis Mealy machine with 6011 states that we subsequently used for model-based testing. Generally, N_{autl} should be chosen as large as possible given the available time budget for testing, as a larger N_{autl} leads to more accurate abstract models.

Test-Case Generation

In the following, we describe random test-case generation for Mealy machines, which serves as a baseline. Then, we discuss three different approaches to model-based test-case generation. Note that our testing goal is to explore the system's state space and to generate system traces with high coverage, with the intention of learning a neural network. Therefore, we generate a fixed number of test cases N_{train} and do not impose conditions on outputs other than those defined by the set *Violations* in the mapper.

Random Testing. The random testing strategy generates input sequences with a length chosen uniformly at random between 1 and the maximum length $l_{\max}$. Inputs in the sequences are also chosen uniformly at random from I. Hence, we generate random words via Algorithm 2.3.

Learning-Based Testing. The learning-based testing strategy relies on active automata learning as described in Chapter 2 and performs active automata learning using the setup introduced in this section via the KV algorithm. It produces exactly those tests that are executed during automata learning and therefore sets N_{autl} to N_{train}. In other words, the traces produced by the learning-based testing strategy

Algorithm 10.1 Output-directed test-case generation

```
Input: $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle, label \in O, N_{\text{train}}$
Output: $TestCases$ : a set of test cases directed to '$label \in O$'
 1: $TestCases \leftarrow \emptyset$
 2: while $|TestCases| < N_{\text{train}}$ do
 3:     $randLen \leftarrow RandomInteger$
 4:     $prefix \leftarrow rSeq(I, randLen)$
 5:     $q_r \leftarrow \delta(q_0, prefix)$
 6:     $q_r' \leftarrow rSel(Q)$
 7:     $interfix \leftarrow PathToState(q_r, q_r')$            ▷ input sequence to $q_r'$
 8:     if $interfix \neq \perp$ then                          ▷ check if path to state exists
 9:         $suffix \leftarrow PathToLabel(q_r, label)$        ▷ input sequence to $label$
10:         if $suffix \neq \perp$ then                        ▷ check if path to label exists
11:             $TestCases \leftarrow TestCases \cup \{prefix \cdot interfix \cdot suffix\}$
12:         end if
13:     end if
14: end while
15: return $TestCases$
```

comprise the first N_{autl} non-cached output queries and test queries performed for automata learning[2]. While this strategy systematically explores the abstract state space of the SUL, it also generates very simple tests during the early rounds of learning, which are not helpful for learning a behaviour model in Section 10.2.2.

Transition-Coverage-Based Testing. The transition-coverage-based testing strategy uses a learned model of the SUL as basis. Basically, we learn a model, fix that model and then generate N_{train} test sequences with the *transition-coverage* testing strategy discussed in Chapter 4 [17]. We use this testing strategy, as it performed well in automata learning and scales to large automata. The intuition behind it is that the combination of variability through randomisation and coverage-guided testing is generally well-suited in a black-box setting.

Recall from Chapter 4 that test-case generation from a Mealy machine $\mathcal{M}$ with this strategy is split into two phases: a generation phase and a selection phase. The generation phase generates a large number of tests by performing random walks on $\mathcal{M}$. In the selection phase, n test cases are selected to optimise the coverage of the transitions of $\mathcal{M}$. Since the n required to cover all transitions may be much smaller than N_{train}, we performed several repetitions, alternating between generation and selection until we selected and executed N_{train} test cases.

Output-Directed Testing. The output-directed testing strategy also combines random walks with coverage-guided testing, but it aims at covering a given abstract output '*label*'. It is based on a learned Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ of the SUL and implemented by Algorithm 10.1. This algorithm generates a set of N_{train} test cases. All test cases consist of a random '*prefix*' that leads to a random source state q_r, an '*interfix*' leading to a randomly chosen destination state q_r' and a '*suffix*' from q_r' to the output '*label*'. The suffix explicitly targets a specific output, the interfix aims to increase the overall SUL coverage and the random prefix introduces variability.

Example 10.6 (Relevant Outputs in Platooning). In the platooning scenario, we aim at covering behaviour relevant to collisions, therefore we generally set $label = crash$ and refer to the corresponding testing strategy also as crash-directed testing. Note that the execution of a crash-

[2]Queries cached by the mapper are not added twice, as they do not provide additional information.

directed test case may not produce an actual crash on the SUL, because we generate test cases from a learned Mealy machine, which may be inaccurate.

10.2.2 Learning a Recurrent Neural Network Behaviour Model

In the considered scenario, we are given length T sequences of vectors $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ with $\mathbf{x}_i \in \mathbb{R}^{d_x}$ representing the concrete inputs to the hybrid system, and the task is to predict corresponding length T sequences of target vectors $\mathbf{T} = (\mathbf{t}_1, \dots, \mathbf{t}_T)$ with $\mathbf{t}_i \in \mathbb{R}^{d_y}$ representing the concrete outputs of the hybrid system. Recurrent neural networks (RNNs) are a popular choice for modelling this kind of problems.

Given a set of N_{train} training input/output sequence pairs $\mathcal{D} = \{(\mathbf{X}_n, \mathbf{T}_n)\}_{n=1}^{N_{\text{train}}}$, the task of machine learning is to find suitable model parameters such that the output sequences $\{\mathbf{Y}_n\}_{n=1}^{N_{\text{train}}}$ computed by the RNN for input sequences $\{\mathbf{X}_n\}_{n=1}^{N_{\text{train}}}$ closely match their corresponding target sequences $\{\mathbf{T}_n\}_{n=1}^{N_{\text{train}}}$. More importantly, the computed output sequences shall generalise well to sequences that are not part of the training set $\mathcal{D}$, that is, the RNN shall produce accurate results on unseen data. To obtain suitable RNN parameters, we typically minimise a loss function describing the misfit between predictions $\mathbf{Y}$ and ground truth targets $\mathbf{T}$. Here, we achieve this through a minimisation procedure known as stochastic gradient descent which works efficiently. For details on RNN learning, we refer to the extended version of our paper on test-based learning of hybrid systems [28].

Example 10.7 (Learning an RNN for Detecting Collisions in Platooning). In our platooning scenario, the inputs $\mathbf{x}_i \in \mathbb{R}^2$ at time step i comprise the input variables U, which include the acceleration value *acc* and the orientation Δ of the leader vehicle in radians. We preprocess the orientation Δ by transforming it to $\Delta' = \Delta_i - \Delta_{i-1}$, the angular difference in radians between orientations of consecutive time steps. By doing that we get rid of discontinuities when these values are constrained to a fixed interval of length 2π. The outputs $\mathbf{y}_i \in \mathbb{R}^3$ at time step i of the hybrid system comprise the values of observable output variables Y. These values include the velocity of the leader v_l, the velocity of the first follower v_f, and the distance d between the leader and the first follower.

Note that RNNs are not constrained to sequences of a fixed length T. However, training with fixed-length sequences is more efficient as it allows full parallelisation through GPU computations. Hence, during test-case execution, we pad sequences at the end with concrete inputs ($acc \mapsto 0, 1$), i.e., the leader drives at constant speed at the end of every test case. In rare cases, the collected test data showed awkward behaviour that needed to be truncated at some time step. This happened, for instance, in cases where the leader's velocity v_l became negative. We padded the affected sequences at the beginning by copying the initial state where all cars have zero velocity. We used this padding procedure to obtain fixed-length sequences with $T = 256$.

In our experiments, we used RNNs with one hidden layer of 100 neurons. Since plain RNNs are well-known to lack the ability to model long-term dependencies, we used long short-term memory (LSTM) cells for the hidden layer [141]. To evaluate the generated training sequences, we trained models for several values of training set sizes N_{train}. We used ADAM [163] implemented in Keras [82] with a learning rate $\eta = 10^{-3}$ to perform stochastic gradient descent for 500 epochs. The number of training sequences per mini-batch was set to $\min\left(\frac{N_{\text{train}}}{100}, 500\right)$. Each experiment has been performed ten times using different random initial parameters and we report average performance values computed from these ten runs.

10.3 Evaluation

In this section, we report on the performance of RNNs trained from datasets obtained through executing test cases produced by the four test-case generation techniques presented in Section 10.2.1. We use two performance indicators: (1) classification into crash-producing and non-crash-producing sequences and (2) prediction of crash time for crash-producing sequences. As a shorthand, we refer to the learning-based testing strategy as LBT, to the transition-coverage-based testing strategy as TCBT, and to output-directed testing as ODT. We perform experiments for varying N_{train} to observe the influence of the training dataset size.

10.3.1 Predicting Crashes with Recurrent Neural Networks

We aim to predict whether a sequence of input values results in a crash, thus we are dealing with a binary classification problem. A sequence is positive, i.e., the sequence results in a crash, if its execution causes leader-follower distance *d* to get below $0.43m$, which is the length of a remote-controlled truck.

For the evaluation, we generated validation sequences with the ODT strategy. This strategy results in sequences that contain crashes more frequently than the other testing strategies which is useful to keep the class imbalance between crash and non-crash sequences in the validation set minimal. We emphasise that these validation sequences do not overlap with the training sequences that were used to train the LSTM-RNN with ODT sequences. The validation set[3] contains $N_{\text{val}} = 86800$ sequences out of which 17092 (19.7%) result in a crash.

For the reported scores of our binary classification task we define the following convenient values:

True Positive (TP): number of positive sequences predicted as positive

False Positive (FP): number of negative sequences predicted as positive

True Negative (TN): number of negative sequences predicted as negative

False Negative (FN): number of positive sequences predicted as negative

We report the following four measures: (1) the classification error (CE) in $\%$, (2) the true positive rate (TPR), (3) the positive predictive value (PPV), and (4) the F1-score (F1). These scores are defined as

$$\text{CE} = \frac{\text{FP} + \text{FN}}{N_{\text{val}}} \times 100 \qquad\qquad \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad\qquad F1 = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

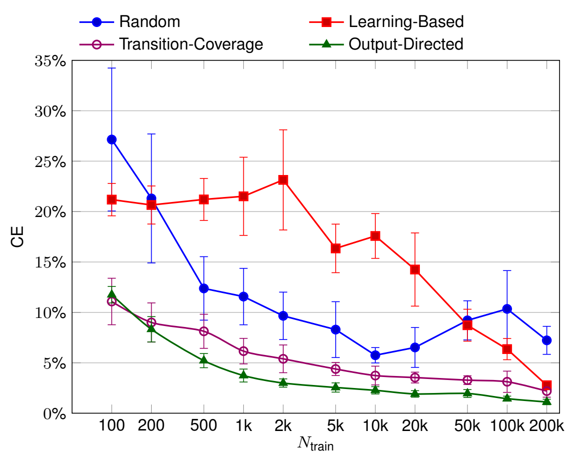
The TPR and the PPV suffer from the unfavourable property that they result in unreasonably high values if the LSTM-RNN simply classifies all sequences as either positive or negative. The F1-score is essentially the harmonic mean of the TPR and the PPV so that these odd cases are ruled out. Note that for the CE a smaller value indicates a better performance, whereas for the other scores TPR, PPV, and F1 a higher score indicates better performance. The CE is always in the range between 0% and 100%, while the other three measures produce values in the interval $[0, 1]$.

The average results and the standard deviations over ten runs for these scores are shown in Figure 10.4. The LSTM-RNNs trained with sequences from random testing and LBT perform poorly on all scores especially if the number of training sequences N_{train} is small. Notably, we found that sequences generated by LBT during early rounds of automata learning are short and do not contain a lot of variability, explaining the poor performance of LBT for low N_{train}.

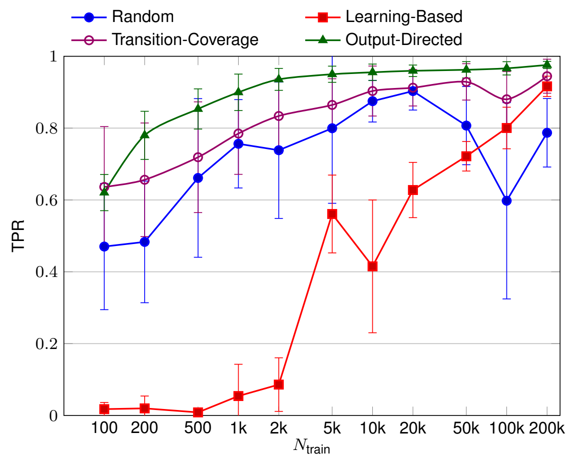
We can observe from the TPR shown in Figure 10.4b that random testing and LBT perform poorly at detecting crashes when they actually occur. Especially the performance drops of LBT at $N_{\text{train}} = 10000$ and of random testing at $N_{\text{train}} = 100000$ indicate that additional training sequences do not necessarily improve the capability to detect crashes, as crashes in these sequences appear to remain outliers.

Training LSTM-RNNs with the TCBT strategy and the ODT strategy outperforms random testing and LBT for all training set sizes N_{train}, where the results slightly favour ODT. The advantage of TCBT

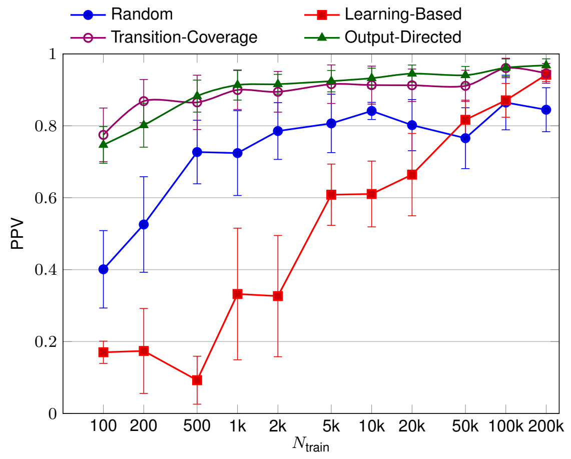
[3]This set is usually called test set in the context of machine learning, but here we adopt the term validation set to avoid confusion with model-based testing.



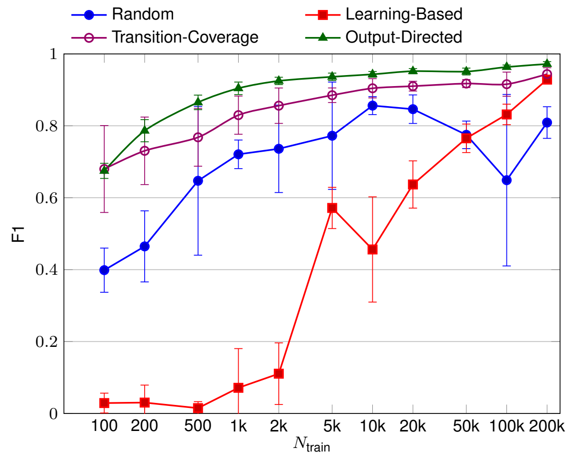
(a) Classification Error (CE)



(b) True Positive Rate (TPR)



(c) Positive Predictive Value (PPV)



(d) F1-score (F1)

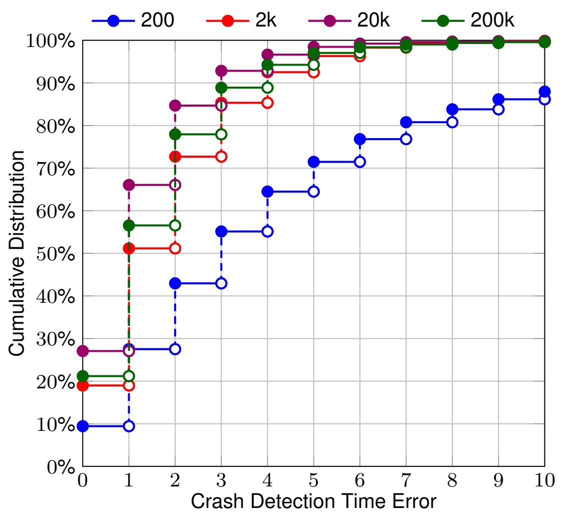
Figure 10.4: Crash detection performance measures for all testing strategies over changing N_{train}.

and ODT becomes evident when comparing the training set size N_{train} required to achieve the performance that random testing achieves using the maximum of $N_{\text{train}} = 200000$ sequences. The CE of random testing at $N_{\text{train}} = 200000$ is 7.23% which LBT outperforms at $N_{\text{train}} = 100000$ with 6.36%, TCBT outperforms at $N_{\text{train}} = 1000$ with 6.16%, and ODT outperforms at $N_{\text{train}} = 500$ with 5.22%. Comparing LBT and ODT, ODT outperforms the 2.77% CE of LBT at $N_{\text{train}} = 200000$ with only $N_{\text{train}} = 5000$ sequences to achieve a 2.55% CE.

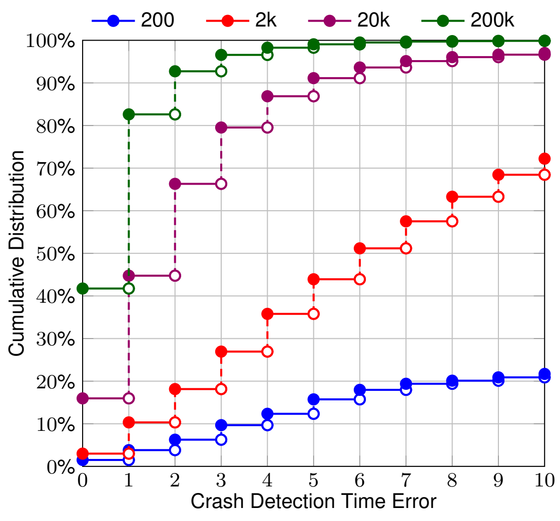
The F1-score is improved similarly: random testing with $N_{\text{train}} = 200000$ achieves 0.809, while TCBT achieves 0.830 using only $N_{\text{train}} = 1000$ sequences and ODT achieves 0.865 using only $N_{\text{train}} = 500$ sequences. Comparing LBT and ODT, LBT achieves 0.929 at $N_{\text{train}} = 200000$ whereas ODT requires only $N_{\text{train}} = 5000$ to achieve an F1-score of 0.936. In total, the sample size efficiencies of TCBT and ODT are two to three orders of magnitudes larger than of random testing and LBT.

10.3.2 Evaluation of the Detected Crash Times

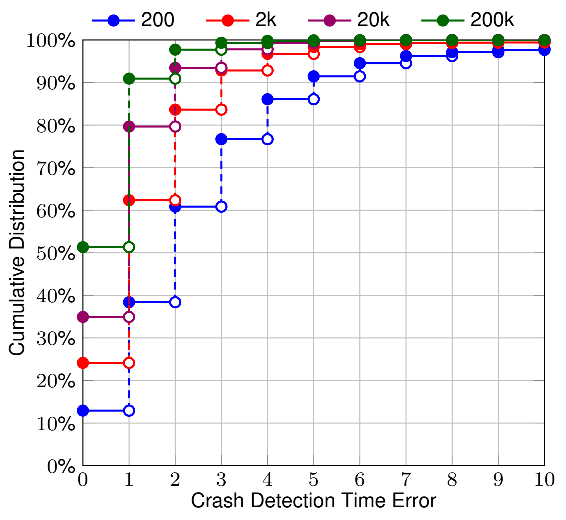
In the next experiment, we evaluate the accuracy of the crash detection time. The predicted crash time is the earliest time step at which d drops below the threshold of $0.43m$, and the crash detection time error is the absolute difference between the ground truth crash time and the predicted crash time. Please note that the crash detection time error is only meaningful for true positive sequences.



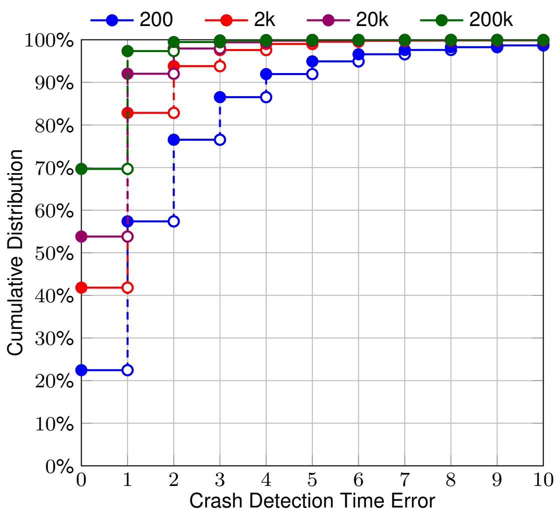
(a) Random testing



(b) Learning-based testing



(c) Transition-coverage-based testing



(d) Output-directed testing

Figure 10.5: This figure shows Cumulative distribution function (CDF) plots for the difference between true crash time and predicted crash time for sequences that are correctly classified as resulting in a crash. Results are shown for all testing strategies and several training dataset sizes N_{train}.

Figure 10.5 shows CDF plots describing how the crash detection time error distributes over the true positive sequences. The x-axis of these plots denotes the crash detection time error e and the y-axis denotes the percentage of traces for which crashes are predicted with an error of at most e. It is desired that the CDF exhibits a steep increase at the beginning which implies that most of the crashes are detected close to the ground truth crash time. The CDF value at crash detection time error 0 indicates the percentage of sequences whose crash is detected without error at the correct time step.

As expected the results get better for larger training sizes N_{train}. Random testing and LBT exhibit large errors and only relatively few sequences are classified without error. With random testing, less than 30% of the crashes in the true positive sequences are classified correctly using the maximum of $N_{\text{train}} = 200000$ sequences. On the other side, TCBT requires only $N_{\text{train}} = 20000$ sequences to classify 34.9% correctly, and ODT requires only $N_{\text{train}} = 2000$ to classify 41.8% correctly. Combining the results from Figure 10.5 with the TPR shown in Figure 10.4b strengthens the crash prediction quality even more. TCBT and ODT do not only achieve a higher TPR, they also predict crash times more accurately. Furthermore, using the maximum of $N_{\text{train}} = 200000$ sequences, TCBT and ODT classify 90.9% and 97.3% of the sequences with an error of at most one time step, respectively.

Discussion. We have observed that LBT and random testing generally perform worse than TCBT and ODT, however, note that TCBT and ODT require a learned automaton as basis. Hence, our experiments demonstrate that automata learning provides us with a model that is sufficiently complete for model-based testing to achieve good coverage of the SUL's state space. It should also be noted that LBT achieves results close to TCBT and ODT at the largest training set size of $N_{\text{train}} = 200000$; see Figure 10.4. This does not hold true for random testing for which we see fluctuating performance.

10.4 Summary

In this chapter, we presented a two-phase process to learn a behaviour model from observations of a hybrid system. The process targets specified behaviour through systematic testing without requiring prior knowledge of the system. We combined abstract active automata learning, model-based testing, and machine learning in our implementation.

Given a black-box hybrid system, we learn an abstract automaton capturing its discretised state space in the beginning of the first process phase. After that, we explore the discretised state space via model-based testing and add variability through randomisation. By testing directed towards the specified behaviour of interest, we are able to cover rare and exceptional behaviour. This results in test suites with high coverage of the targeted behaviour from which we generate behavioural datasets.

In the second phase of the process, we train LSTM-RNNs on the behavioural datasets to learn behaviour models. A real-world case study with a platooning scenario demonstrates the advantage of our approach over alternative data generation via random sampling. Experimental evaluations show that LSTM-RNNs learned with model-based data generation achieved significantly better results compared to models learned from randomly generated data. For instance, it was possible to reduce the classification error by a factor of five. Model-based data generation also improved the sample size efficiency. It required up to three orders of magnitude fewer training samples than random testing to achieve a similar F1-score.

10.5 Results and Findings

We will now briefly discuss our results in the context of the research questions defined in Section 1.6.3. Note that hybrid automata basically generalise timed automata, as explained in Section 10.1.

RQ 1.2 What guarantees can be given if randomised testing is applied? Unfortunately, we cannot provide strong guarantees on the outcome of the proposed learning process. We cannot guarantee that the learned models converge to the true model. However, we have empirically shown that our data-generation process is able to generate training data that enables learning of accurate RNN models. In Figure 10.4, we see that the error bars on the graphs for our model-based data generation are small, thus we reliably produced accurate models. Although we did not learn perfectly "correct" models, we want to address **RQ 1.3** with that. This research question asks whether correct models can be learned reliably by applying randomised testing.

The proposed testing approach could also be used for learning-based testing, that is, to search for errors such as safety violations. This has been demonstrated by Meinke in a platooning scenario [205]. The form of abstraction that we apply ensures that learning-based testing does not produce false positives with respect to reachability[4]. Hence, we can guarantee that if we detect a collision, then a collision is

[4]As in Section 10.3, a positive outcome corresponds to reaching the behaviour of interest.

indeed possible. If a collision is reachable in a learned hypothesis automaton, then it is reachable in the black-box hybrid system.

RQ 3.2 What assumptions are sufficient to enable learning in the context of real-time systems? The test-based learning approach presented in this chapter requires very few assumptions. Essentially, we require the existence of a set of controllable input variables and two sets of observable variables. Strictly speaking, there must exist a suitable discretisation of the input space for our approach to be applicable.

Concluding Remarks. We were able to apply learning-based testing using the transition-coverage-based testing strategy for a hybrid system with very large state space, although the testing strategy has been developed for medium-sized discrete systems in Chapter 4. This demonstrates a certain degree of generality of learning-based testing. With output-directed testing, we applied a similar testing strategy as for reachability in stochastic systems; see Chapter 6. Despite differences on the surface, the work presented in this chapter applies techniques that we discussed in previous chapters.

It is also noteworthy that we found only very few collisions in initial experiments without using the $const_l$ input. In particular, we did not find any collisions during random testing. Upon analysing sequences leading to collisions that we found in early automata-learning experiments, we added the $const_l$ input. This increased the frequency of collisions across all test-case generation techniques. Hence, learning-based testing can aid manual analysis and vice versa.

11 Related Work

Declaration of Sources

We discussed related work in each of the publications that form the basis for this thesis, therefore this chapter mainly follows the presentation of related work in our publications [15, 16, 17, 18, 28, 28, 29, 263, 264, 265, 267], with the exception of the first two sections. These two sections are mainly based on our survey on model learning and model-based testing [26] and on the extended article on fault-based testing in learning [17].

This chapter discusses work related to the research presented within this thesis. Since the main focus of this thesis is on learning-based testing and test-based learning, the first two sections consider these two lines of research. Section 11.1 gives an overview of works that use model learning for model-based testing, while Section 11.2 considers the other direction by discussing work that uses model-based testing to enable model learning. Section 11.3 discusses learning of stochastic and non-deterministic system models, whereas Section 11.4 discusses learning of real-time system models.

Note that we explicitly focus on models of software systems. Automata learning has traditionally been applied for grammatical inference, that is, to learn hidden language representations based on data such as text [95]. In fact, L^* as the most influential active automata learning algorithm targets the problem of learning automata accepting some regular language [37]. Consequently, there is a substantial amount of research on automata learning in grammatical inference and more generally in computational linguistics that we cannot thoroughly cover in this chapter. However, we will comment on work that is closely related to this thesis and on influential work in general. For more information, we refer to the excellent book by Colin de la Higuera [95].

We also identified related research that does not consider traditional automata-learning problems. Probabilistic black-box reachability checking presented in Chapter 6 addresses simulation-based verification of stochastic systems. Section 11.5 discusses work related to that. We learned real-time system models from test observations via genetic programming, therefore we discuss the application of meta-heuristics in model learning and testing in Section 11.6. Finally, Section 11.7 covers related work that does not fit into any of the main categories. The corresponding research areas, which are not in the focus of this thesis, include verification of platooning systems, model-based protocol testing, model-based mutation testing, and testing of real-time systems.

11.1 Model Learning for Model-Based Testing

In the following, we will review works in the area of learning-based testing. Works in this area apply learning to enable or to support testing. There are various approaches to that and most of the works discussed below have in common that they consider stateful systems. This section is mainly based on Section 5 and Section 6 of our survey [26]. Therefore, it groups the approaches similarly.

11.1.1 Conformance Testing

We discussed the basic form of conformance testing in Section 2.4. The goal of conformance testing is to test for conformance between a SUT and a specification model. Typically behavioural conformance testing is combined with model learning by learning specification test models to use them subsequently for generating a conformance test suite [2, 283].

Aarts et al. [4, 5] explore an alternative way to integrate learning into testing. The authors propose to learn a model of both a reference implementation and the implementation under test and then use equivalence checking tools to check for equivalence between the two learned models. This way conformance testing is performed in an intensional manner by comparing models rather than by generating test cases from the specification model and executing test cases on the implementation.

This approach is similar to our MQTT case study presented in Chapter 3 [263]. In this case study, we learned models of different implementations and compared them among each other. Detected differences are considered to point to potential bugs that should be analysed manually. The system HVLEARN described by Sivakorn et al. [253] follows a similar approach as well. It learns DFA models of SSL/TLS hostname verification implementations via the KV algorithm [160]. Given learned models, HVLEARN is able to list unique differences between pairs of models and additionally provides analysis capabilities for single models. The authors reported that they found eight unique, previously unknown RFC violations by comparing learned models. Another example using a similar technique in the security domain is SFADIFF [39]. In contrast to the other approaches, it learns symbolic finite automata (SFA) and is able to find differences between pairs of sets of programs. This can, for instance, be used for fingerprinting or for creating evasion attacks against security measures. It has been evaluated in case studies involving TCP state machines, web application firewalls, and parsers in web browsers.

These approaches to conformance testing between implementations generally cannot guarantee exhaustiveness. In other words, if models are found to be equivalent this does neither imply that the implementations are equivalent nor that the implementations are free of errors. In testing of complex systems, however, the reverse will often hold: it is likely that differences can be detected. These differences may either help to extend the learned models in case learning introduced the differences, or they may be actual differences between the considered systems. The discussed case studies showed that found differences can be exploited in practice. For instance, we detected bugs in MQTT by analysing differences.

11.1.2 Requirements-Based Testing

With the introduction of black-box checking, Peled et al. [229, 230] pioneered a line of research combining learning, black-box testing, and formal verification. In order to check whether a black-box system satisfies some formally-defined property, a model is learned with Angluin's L^*-algorithm and the property is checked on this model. If a counterexample is found, it either shows that the property is violated or it is spurious and can be used to extend the model. To avoid false positives, conformance testing via the W-method [83, 277] is used to extend the model, i.e., the W-method is used to implement equivalence queries. Black-box checking work also inspired our probabilistic black-box reachability checking presented in Chapter 6.

Following that, several optimisations and variations have been proposed. Adaptive model checking [127, 128, 129] optimises black-box checking by using a model of the system which is assumed to be inaccurate but relevant. Another early extension is grey-box checking [103], which considers a setting in which a system is composed of some completely specified components and some black-box components. With regard to testing, the W-method [83, 277] and other conformance testing approaches, taking the grey-box setting into account, are used and compared.

Adaptive model-checking combined with assume-guarantee verification has been considered for the verification of composed systems [148]. Furthermore, another variation of adaptive model-checking has been described by Lai et al. [175]. The authors apply genetic algorithms instead of L^* to learn a system model. Their results show promising performance for prefix-closed languages.

Meinke and Sindhu [206] applied the learning-based testing paradigm to reactive systems and presented an incremental learning algorithm for Kripke structures. In their approach, an intermediate learned model is model-checked against a temporal specification in order to produce a counterexample input stimulus. The SUT is then tested with this input. If the resulting output satisfies the specification, then this new input-output pair is integrated into the model. Otherwise, a fault has been found and the algorithm terminates. Meinke and Sindhu also implemented the tool LBTEST which supports learning-based testing of reactive systems with respect to properties expressed in propositional linear temporal logic [207].

There are various approaches to automatically infer non-automata-based specifications for testing. While a complete discussion is beyond the scope of this section, we want to name two such approaches. TORADOCU, for instance, generates test oracles for exceptional behaviour of procedures via natural languages processing of tagged Java code comments [121]. JDOCTOR similarly analyses code comments by applying natural languages processing, but supports more tags and covers not only exceptional behaviour, but also preconditions and non-exceptional postconditions [61]. The authors of TORADOCU and JDOCTOR note that they plan to also support temporal properties, for instance about call protocols, in the future [62]. Automata-learning techniques could be applied for this purpose [35].

11.1.3 Security Testing

Based on black-box checking [230], Shu and Lee described an approach to learning-based security testing [250]. Instead of checking more general properties, they try to find violations of security properties in the composition of learned models of components. In the following work, they presented a combination of learning and model-based fuzz testing and considered both active and passive model learning [251]. This approach is more extensively described by Hsu et al. [146] with a focus on passive model learning. For this purpose, the authors detail their state-merging-based learning approach and discuss the type of fuzzing functions and the coverage criteria they used. Additionally, they provide a more extensive evaluation.

A compositional approach is also followed by Oostdijk et al. [226] by using various methods to study the security of cryptographic protocols, where learning by testing black-box implementations is one of the techniques employed. The secrecy and authenticity properties are then checked on both the protocol specifications and the actual implementations through the learned model of the implementation.

Hossen et al. [143] presented an approach to model learning specifically tailored to security testing of web applications. The approach is based on the Z-quotient algorithm [232].

Cho et al. [80] developed a security testing tool called MACE. This tool combines the learning of a Mealy machine with concolic execution of the source code to explore the state space of protocol implementations more efficiently. Here, the learning algorithm guides the concolic execution in order to gain more control over the search process. When applied to four server applications, MACE could detect seven vulnerabilities.

De Ruiter and Poll [97] learned models of implementations of the TLS protocol via L^* [37] using a similar learning setup as we used in Chapter 3. After learning, they manually investigated the learned models, specifically targeting security-related flaws. They refer to their approach as protocol state fuzzing, as active automata learning requires fuzzing of various message sequences.

11.1.4 Integration Testing

Tackling the issue that complex systems commonly integrate third-party components without any specification, Li, Groz and Shahbaz [190] proposed a learning-based approach to integration testing. They follow an integrated approach in which they learn models of components from tests and based on the composition of these models, they generate integration test cases. The execution of these test cases may eventually lead to an update of the learned models if discrepancies are detected. Integration testing thus serves also as equivalence oracle. In following work, Li et al. [190, 248, 249] extended their learning-based integration testing approach to more expressive models. These models also account for data through the introduction of parameters for actions and predicates over input parameters. Additionally, they also allow for observable non-determinism [248, 249].

Groz et al. [131] present an alternative approach to learning of component models. Instead of learning each component model separately, they infer a *k-quotient* of the composed system and learn component models by projection. With an initial model at hand, they perform a reachability analysis to detect compositional problems. If a detected problem can be confirmed, they warn that a problem exists, otherwise they refine the inferred models if the problem could not be confirmed. Testing is stopped once no new potential compositional problem can be found.

In a setting similar to that considered by Li et al. [190] and using the same algorithm, Shahbaz et al. [249] describe an approach to detect feature interaction in an integrated system. Basically, they learn models of components by testing and execute the same test cases on the composed system again. If the observations in the second phase do not conform to the observations on the learned models, a feature interaction is detected.

Based on their previous works, Shahbaz and Groz [247] present an approach for analysing and testing black-box components by combining model learning and model-based testing techniques. The proposed procedure starts by learning each component's (partial) behavioural model and composing them to form a product. The product is then fed into a model-based test-case generator. The generated test cases are subsequently executed on the real system. Discrepancies between the learned models and the system's observed behaviour form the basis to refine the learned models.

Kunze et al. [171] devised a test-based learning approach, where an *already specified system* under test is executed to find and record deviations from the known specification. Based on the collection of these deviations, a fault model is learned. This fault model is then used to perform model-based testing with QuickCheck [40, 84] to discover similar faults in other implementations. Being a preliminary work, it learns abstract deterministic Mealy machines via LearnLib [152], that is, it does not consider data during learning. The models in this approach are generally rich state-based models with full support for predicates. It falls into the integration-testing category in that the overall goal of the work is to test implementations composed of different versions of components, some of which may exhibit deviations from the reference model.

11.1.5 Regression Testing

Hagerer et al. [134] and Hungar et al. [149] consider regression testing as a particularly fruitful application scenario for model learning. The possibility of automatically maintaining models during the evolution of a system could greatly improve and aid regression testing.

Schuts et al. [244] discuss an industrial application of model learning to support refactoring of legacy software. Similarly to our MQTT case study and the work by Aarts et al. [4, 5], they learn models of two different implementations of the same application; in this case, the models of legacy software and refactored software. The learned models are then checked for equivalence to determine whether the examined software systems behave equivalently. Differences may reveal errors in both the legacy and the refactored system.

11.1.6 Performance Testing

Adamis et al. [9] proposed an approach to passively learn FSM models from conformance test logs for performance testing. The approach aids performance testing by providing test models for systems with unknown structure, which can be used as a basis for load testing. Since the learned models may be inaccurate, manual postprocessing is required.

Aichernig and Schumi learned timing information from logs of MQTT brokers and augmented existing functional models with this information [13, 30]. Furthermore, they evaluated the performance of the considered MQTT brokers through a statistical analysis of the extended models; see also Section 11.4.

11.1.7 GUI Testing

Choi et al. [81] presented SWIFTHAND, a passive-learning-based testing tool for user interfaces of Android apps. The authors interleave learning and testing as follows: (1) they use a learned model to steer testing towards previously unexplored states and (2) refine learned models based on test observations. Their test-case selection strategy aims at minimising the number of restarts, the most time-consuming action in the considered domain, while maximising (code) coverage. The evaluation shows that SWIFTHAND outperforms L^*-based and random testing.

Mariani et al. [201, 202] presented AUTOBLACKTEST, a technique and tool for the automatic generation of system-level test cases for GUI testing. The authors propose to perform Q-learning via testing to automatically and incrementally learn models of GUI-based applications and how to interact with them. An experimental comparison between AUTOBLACKTEST and GUITAR demonstrates that AUTOBLACKTEST achieves a higher code coverage and is more effective in terms of fault detection.

11.1.8 Protocol Testing

Learning-based testing in networked environments, which is the focus of this thesis, is generally a popular application area. Various security protocols and protocol implementations have been analysed through learning-based approaches. Applying the abstraction technology described by Aarts et al. [3] and LearnLib [152], Fiterău-Broştean et al. [111, 112] report on learning Mealy machine models of different TCP stack implementations. The authors did not use the models directly for testing, though. They applied model checking to verify properties of the composition of client and server implementations in an off-line fashion [112]. Similarly to black-box checking [230], counterexamples returned from the model checker can be considered test cases that demonstrate invalid behaviour, unless the checked model had been learned incorrectly. Fiterău-Broştean et al. [113] carried out a similar case study in a security setting focused on SSH implementations. Model-checking the learned models of different implementations revealed minor violations of the standard but no security-critical issues. Walkinshaw et al. [287] applied their inductive testing approach to explore the behaviour of the Linux TCP stack. Aarts, Kuppens, Tretmans, Vaandrager and Verwer [4, 5] combined various techniques to learn and test the bounded retransmission protocol.

Comparetti et al. [89] use learned protocol models as an input for fuzzing tools in order to discover security vulnerabilities. They learned a number of malware, text-based and binary protocols using domain-specific and heuristics-based learning techniques. Learning-based fuzz testing has also been applied for the Microsoft MSN instant messaging protocol [146, 251] (see also requirement-based testing above). Furthermore, learning-based testing of security protocols has been addressed by Shu and Lee [250] as well. As mentioned in Section 11.1.3, de Ruiter and Poll [97] learned models of implementations of the TLS protocols, which they analysed to detect security-related flaws.

11.1.9 Web Service Testing

Raffelt et al. [237] applied dynamic testing on web applications. More concretely, they described a test environment, called WEBTEST, which combines traditional testing methods, like record-and-replay, and dynamic testing. The latter provides benefits such as systematic exploration and model learning, while the former eases dynamic testing by defining possible input actions.

11.2 Model-Based Testing for Model Learning

The Java library LearnLib [152] implements standard conformance testing algorithms for active automata learning. These standard algorithms include the W-method [83, 277] and the partial W-method [117] that we discussed in Section 2.4. Early work proposing the application of the W-method in combination with active automata learning was black-box checking [230]. More recently, a slightly modified version of the W-method has been applied for learning models of TLS servers [97].

The Zulu challenge [88] addressed the problem of implementing automata learning without equivalence queries and a limited number of membership queries. Put differently, it called for solutions to test-based automata learning with a limited testing budget. Howar et al. reviewed their experience gained in this challenges and noted that it is necessary to find counterexamples to equivalence with few tests for automata learning to be practically applicable [144]. Similar to our fault-based testing approach, they described a testing heuristic inspired by Rivest and Schapire's counterexample processing [240]. This is in contrast to deterministic conformance testing via the W(p)-method that guarantees conformance relative to some bound on the number of states.

Smeenk et al. [254] applied a partly randomised conformance testing technique, similar to the random W-method discussed in Section 2.4.2. In order to keep the number of tests small, they aimed to test effectively by determining adaptive distinguishing sequences using a technique described by Lee and Yannakakis [183]. With this technique and domain-specific knowledge, they succeeded in learning a large model of industrial control software. The same technique has also been used to learn models of TCP implementations [112] and SSH implementations [113].

Chapter 4 presents two evaluations of active automata learning configurations with respect to efficiency in terms of required testing budget. Berg et al. [55] performed early work on the practical evaluation of the performance of L^*-based learning. The authors studied the impact of various system properties on the learning performance, including the alphabet size, the number of states, and prefix-closedness of the target language. Smetsers et al. [255] presented an efficient method for finding counterexamples in active automata learning that applies mutation-based fuzzing. In their evaluation, they compared four learning configurations with respect to learning performance in terms of the size of learned models and the required number of queries. They considered combinations of the L^* algorithm [37] and the TTT algorithm [151] with their proposed method and the W-method [83, 277].

Groz, Brémond, and Simão applied concepts from finite-state-machine-based testing to implement an efficient algorithm for active learning of Mealy machines without resets [70, 132]. In particular, their algorithm uses (adaptive) homing sequences and characterising sets to learn Mealy machines from a single system trace that continually grows during learning. Moreover, the authors evaluated various active learning configurations, including L^*-based configurations with and without resets.

11.3 Learning Models of Stochastic and Non-deterministic Systems

In Chapter 6, we presented a testing technique that is based on stochastic automata learning and in Chapter 7, we presented an L^*-based approach to stochastic automata learning. Therefore, we will review related work on stochastic automata learning in this section.

Most sampling-based learning algorithms for stochastic systems are passive, hence they assume pre-existing samples of system traces. Their roots can be found in grammatical inference techniques, such as ALERGIA [74] and RLIPS [75], which identify stochastic regular languages. Similarly to these techniques, we also apply Hoeffding bounds [142] in L^*_{MDP} to test for difference between probability distributions.

Mao et al. [197, 198, 199] learned stochastic automata models with the purpose of verification. More concretely, they learned models for model checking. Notably, they developed IOALERGIA as an extension of ALERGIA to learn MDPs [198, 199]. This learning technique basically creates a tree-shaped representation of the sampled system traces and repeatedly merges compatible nodes to create an automaton. Finally, transition probabilities are estimated from observed output frequencies. Like L^*_{MDP}, IOALERGIA converges in the limit, but showed worse accuracy in evaluation experiments. We applied the passive learning technique IOALERGIA in an active setting in Chapter 6 without changing the learning algorithm itself. Chen and Nielsen [79] also described active learning of MDPs based on IOALERGIA. They proposed to generate new samples by directing sampling towards uncertainties in the data as a way to reduce the number of traces required for learning. In our active application of IOALERGIA, we direct sampling towards parts of the system that are relevant to reachability properties. In L^*_{MDP}, our active learning algorithm for MDPs, we base the sampling strategy not only on the data collected so far (refine queries), but also on the current observation table and the derived hypothesis MDPs (refine & equivalence queries). With that, we do not only target uncertainties in the collected data, but we also take information about the SUL's structure into account.

Wang et al. [289] apply a variant of ALERGIA as well. They take properties into account during model learning with the goal of probabilistic model-checking. More specifically, they apply automated property-specific abstraction/refinement to decrease the model-checking runtime. Nouri et al. [221] also combine stochastic learning and abstraction with respect to some property. Their goal is to improve the runtime of SMC. Notably, their approach could also be applied for black-box systems, similar to probabilistic black-box reachability which applies SMC as well. However, they do not consider systems that are controllable via inputs. Further work on SMC of black-box systems can be found in [245, 294].

Ghezzi et al. [118] presented the BEAR approach that combines inference of probabilistic models and probabilistic model-checking to analyse user behaviour in web applications. More concretely, the authors infer labelled discrete-time Markov chains extended with rewards which they analyse using PRISM [174]. In contrast to our work, they assume all states to be distinguishable through the labelling function, i.e., two different states cannot have the same label. The proposed approach, for instance, enables the detection of navigational anomalies.

L^*_{MDP} builds upon Angluin's L^* [37], therefore it shares similarities with other L^*-based work such as active learning of Mealy machines [200, 246]. Interpreting MDPs as functions from test sequences to output distributions is similar to the interpretation of Mealy machines as functions from input sequences to outputs [258].

Volpato and Tretmans presented an L^*-based technique for non-deterministic input-output transition systems [283]. They simultaneously learn an over- and an under-approximation of the SUL with respect to the **ioco** relation [270], a popular conformance relation in conformance testing. Inspired by that, L^*_{MDP} uses completeness queries and adds transitions to a chaos state in case of low/incomplete information.

Beyond that, we consider systems to behave stochastically rather than non-deterministically. While Volpato and Tretmans [283] leave the concrete implementation of queries unspecified, L^*_{MDP}'s implementation closely follows Section 7.4. Early work on **ioco**-based learning for non-deterministic systems has been presented by Willemse [293]. Khalili and Tacchella [161] addressed non-determinism by presenting an L^*-based algorithm for non-deterministic Mealy machines. As Volpato and Tretmans [283], they assume to be able to observe all possible outputs produced in response to input sequences through the repeated application of these sequences. Their stochastic interpretation of SULs to implement queries motivated our research on stochastic model learning. Both learning approaches for non-deterministic models assume a testing context, as we do.

Related to L^*_{MDP}, L^*-based learning for probabilistic systems has also been presented by Feng et al. [106]. They learn assumptions in the form of probabilistic finite automata for compositional verification of probabilistic systems in the assume-guarantee framework. Their learning algorithm requires queries returning exact probabilities, hence it is not directly applicable in a sampling-based setting. The learning algorithm shares similarities with an L^*-based algorithm for learning multiplicity automata [56], a generalisation of deterministic automata. In order to extend the applicability of learning-based assume-guarantee reasoning for probabilistic systems, Komuravelli et al. [165] presented techniques for learning labelled probabilistic transition systems from stochastic tree samples, where they rely solely on equivalence queries for active learning. Moreover, they described how their algorithms can be applied to generate assumptions for assume-guarantee reasoning. Bouchekir and Boukala also consider assume-guarantee reasoning and proposed an active algorithm for learning assumptions in the form of symbolic interval Markov decision processes [67]. The authors note that they adapted the L^* algorithm, but their learning algorithm works directly on assumptions and uses only equivalence queries. For this reason, the author of this thesis fails to see the relation to L^*-based learning. Their implementation of equivalence queries applies membership queries internally, though.

Further query-based learning in a probabilistic setting has been presented by Tzeng [272]. The author described a query-based algorithm for learning probabilistic automata and an adaptation of Angluin's L^* for learning Markov chains. In contrast to our exact learning algorithm $L^*_{\text{MDP}^e}$, which relies on output distribution queries, Tzeng's algorithm for Markov chains queries the generating probabilities of strings. Castro and Gavaldà review passive learning techniques for probabilistic automata with a focus on convergence guarantees and present them in a query framework [77]. Unlike MDPs, the learned automata cannot be controlled by inputs.

11.4 Learning Models of Real-Time Systems

Work closely related to our work on learning timed systems with respect to the domain has been performed by Verwer et al. [279]. They passively learn deterministic *real-time automata* via state-merging from negative and positive data. In addition to pure state-merging, they also split states and transitions depending on the given data. Deterministic real-time automata measure the time between two consecutive events and use guards in the form of intervals, therefore they have a single clock which is reset on every transition. Verwer et al. do not distinguish between inputs and outputs. Hence, they do not assume output urgency, however, they also do not consider systems to be input enabled, as required in a testing context. They adapted their approach to learn solely from positive data [280]. This adaptation learns probabilistic real-time automata by applying likelihood-ratio tests to decide if merge operations or split operations should be performed. Improvements of this adaptation [280] have been presented by Mediouni et al. [204].

Similarly, Mao et al. applied state-merging to learn *continuous time Markov chains* [199]. This kind of Markov chains defines an exponentially distributed sojourn time for each state. A state-merging-based learning algorithm for more general stochastic timed systems has been proposed by de Matos Pedro et al. [96]. They target learning *generalised semi-Markov processes*, which are generated by stochastic timed automata. All these techniques have in common that they consider systems where the relation between events is fully described by a system's structure. Pastore et al. [227] learn specifications capturing the duration of (nested) operations in software systems. A timed trace therefore includes the start and end of each operation, thus a trace records pairs of related events. Their algorithm, called *Timed k-Tail*, is based on the passive learning technique *k-Tail* [60], which the authors extended to handle timing aspects.

Grinchtein et al. [125, 126] developed active learning approaches for deterministic *event-recording automata* [34], a subclass of TA with one clock per action. The clock corresponding to an action is reset upon its execution essentially recording the time since the action has occurred. While the expressiveness of these automata suffices for many applications, the runtime complexity of the active learning techniques is high and may be prohibitive in practice. Currently, there is no implementation to actually measure runtime. Furthermore, event-recording automata cannot model certain timing patterns. For instance, it may not be appropriate to always reset a clock when considering input-enabled systems. Wei-Lin et al. [191] also presented an active learning algorithm for event-recording automata and applied it to learn assumptions in compositional verification via assume-guarantee reasoning [192]. However, they considered a white-box setting.

Jonsson and Vaandrager [157] note that the active learning approaches discussed above [125, 126] are complex and developed a more practical active-learning approach for Mealy machines with timers. Currently, there is no implementation for this approach as well. A further drawback is that input edges cannot be restricted via guards. They can only model timeout that cause outputs to be produced.

Schumi et al. [27, 243] also learned the timing behaviour of reactive systems. In contrast to the abovementioned approaches, they did not learn the system structure, but augmented known discrete-time system models with timing information that they learned via multiple linear regression. They used the augmented models to statistically analyse the response times of web service applications. In subsquent work, Aichernig, Kann and Schumi [25] used the proposed statistical analysis technique to examine different deployments of the same system with respect to response times. Aichernig and Schumi applied this technique in an IoT context by determining the message latency of MQTT communications [13]. Their work complements the work presented in Chapter 3, as they focus on non-functional behaviour, while we check MQTT implementations for functional correctness. More recently, Aichernig et al. [30] improved the analysis accuracy by applying deep learning.

Nenzi et al. [215] mine specifications in signal temporal logic (STL) which distinguish between regular and anomalous time-dependent system behaviour. An important difference to our work is that they perform a classification task, while we learn models producing the same traces as the systems under consideration.

11.5 Strategy Generation for Stochastic Systems

In Chapter 6, we aim to find optimal schedulers for MDPs, in order to use them as testing strategies. Variations of this problem have been tackled in other simulation-based verification approaches, such as SMC, as well. A lightweight approach for finding schedulers in SMC has been described by Legay et al. [90, 188]. By representing schedulers efficiently, they are able to consider history-dependent schedulers. Through "smart sampling" they achieve to find near-optimal schedulers with low simulation budget. Brázdil et al. [69] presented an approach to unbounded reachability analysis via SMC. The technique is based on delayed Q-learning, a form of reinforcement learning, and requires only limited knowledge of the system. However, it requires more knowledge than the technique discussed in Chapter 6. Another approach using reinforcement learning for strategy generation for reachability objectives has been presented by David et al. [93]. They minimise expected cost while respecting worst-case time bounds.

Learning-based synthesis of control strategies for MDPs has also been studied by Fu and Topcu [116]. They obtain control strategies which are approximately optimal with respect to linear temporal logic (LTL) specifications. They consider transition probabilities to be initially unknown, but in contrast to our setting, they assume the MDP structure to be known.

11.6 Metaheuristic Approaches to Model Learning and Testing

Metaheuristic search as an alternative to classical automata learning such as Angluin's L^* [37] has been proposed by Lai et al. for finite state machines [175]. They apply genetic algorithms and assume the number of states to be known. Lucas and Reynolds compared (evidence driven) state merging with an evolutionary algorithm, while also fixing the number of states for runs of the latter [194]. Their proposed evolutionary algorithm achieved better accuracy than the state-merging-based approach in experiments with small target automata. Additionally, they evaluated a genetic-programming-based technique for automata of variable size and found that this technique performs worse than their evolutionary algorithm.

Lefticaru et al. similarly assume the number of states to be known to generate state machine models via genetic algorithms [186]. Their goal is to synthesise a model satisfying a specification given in temporal logics, rather than learning a model of a black-box system. Early work suggesting a similar approach has been performed by Johnson [156]. A similarity to our metaheuristic learning technique is that Johnson does not require the solution size to be known. Instead, he allows automata to grow via mutation. In contrast to our work, Johnson does not apply crossover, noting that it is not clear how to perform this kind of operation for automata. Genetic-programming-based synthesis by Katz and Peled [159] aims at generating a correct program or model on the source code level. They successfully synthesised solutions to the mutual exclusion problem and applied their technique to locate and correct errors in a communication protocol.

Evolutionary methods have been combined with testing in several areas: Abdessalem et al. [8] use evolutionary algorithms for the generation of test scenarios and learn decision trees to identify critical scenarios. Using the learned trees, they can steer the test-case generation towards critical scenarios. The tool EVOSUITE by Fraser and Arcuri [115] uses genetic operators for optimising whole test suites at once, increasing the overall coverage, while reducing the size of the test suite. Walkinshaw and Fraser presented *Test by Committee*, which is test-case generation using uncertainty sampling [285]. The approach is independent of the type of model that is inferred and an adaption of *Query By Committee*, a technique commonly used in active learning. In their implementation, they infer several hypotheses at each stage via genetic programming, generate random test cases and select those test cases which lead to the most disagreement between the hypotheses. In contrast to most other considered works, their technique does not infer state-based models. It rather infers functions mapping from numerical inputs to single outputs.

We mentioned the work by Nenzi et al. [215] also in Section 11.4, as they mine specifications of time-dependent behaviour. They implemented that by applying an evolutionary algorithm to learn the structure of STL properties.

11.7 Further Related Work

Verification of Platooning Systems. In the work presented in Chapter 10, we performed test-based learning and learning-based testing of a platooning system [29]. Meinke [205] used learning-based testing to analyse vehicle platooning systems with respect to qualitative safety properties, such as collisions. While the automata learning setup is similar to our approach, the author focused on different aspects. Meinke aimed to analyse (1) how well a multi-core implementation of learning-based testing scales and (2) how problem size and other factors affect scalability, whereas we analysed the quality of the training data generated for machine learning. Fermi et al. [107] applied rule inference methods to validate collision avoidance in platooning. More specifically, they used decision trees as classifiers for safe and unsafe platooning conditions and they suggested three approaches to minimise the number of false negatives. Rashid et al. [239] modelled a generalised platooning controller formally in higher-order logic. They proved satisfaction of stability constraints in HOL LIGHT and showed how stability theorems can be used to develop runtime monitors.

Larsen et al. [180] demonstrated how to synthesise safe and optimal controllers for adaptive cruise control involving two cars. Hence, they consider a similar setting as Chapter 10, but they have an opposing goal. While we assume a given controller for the follower and search for unsafe traces via learning-based testing, they generate safe and optimal controllers for the follower using UPPAAL STRATEGO. In more recent work on control synthesis with guaranteed safety, they illustrate the proposed synthesis method on the same adaptive cruise control scenario [182].

Model-Based Protocol Testing. In the following, we will discuss selected works in the area of model-based protocol testing. For a thorough technical overview and review of methods for protocol conformance testing, we refer to Bochmann and Petrenko [284]. Lee and Yannakakis also cover protocol conformance testing in their survey on testing based on finite state machines [184]. While learning-based testing can be considered to be a special form of model-based testing, we will not discuss learning-based approaches. Such approaches are covered in Section 11.1.

Weiglhofer et al. applied fault-based conformance testing on the SIP protocol [291]. They modelled the protocol using LOTOS and followed a mutation-based approach for test-case generation. Belinfante et al. [53] also applied LOTOS for creating formal test models, i.e. specifications, of a protocol. In addition to LOTOS, they also used PROMELA and SDL to model the conference protocol. Furthermore, they introduced the TORX environment for testing, the predecessor of JTORX [51]. Botincan and Novakovic tested the conference protocol using Spec Explorer [66], a model-based testing tool for the .NET platform that supports textual modelling.

Jard and Jéron present the TGV tool [154], a tool for testing based on transition systems. The model-based testing approach implemented by the tool has been validated in various experiments including the conference protocol and other protocols. Fernandez et al. [109] presented a preliminary version of TGV.

Beurdouche et al. [57, 58] tested the TLS protocol, while targeting state-machine flaws, as we did in our case study on learning-based testing of the MQTT protocol. They followed a test-based approach, but generated test cases from a known model via some heuristics. With that they checked for specific faults, such as faults related to skipping mandatory steps in a protocol. Our learning-based approach allows to detect this kind of faults as well.

Model-Based Mutation Testing. In our fault-based approach to efficient test-case generation for active automata learning presented in Chapter 4, we combine model-based mutation testing and random testing. Mutation testing was originally proposed to assess the adequacy of test suites [99]. In contrast, model-based mutation testing is a fault-based test-case generation technique [11, 22, 23]. It injects faults, called mutations, into a specification model and generates test cases that cover those faults. Executing these test cases basically allows to show that certain faults have not been implemented. Early work in this area has been performed by Budd and Gopal [72] who mutated specifications given in predicate calculus. An example of early work considering automata-based specification models has been presented by Aichernig and Corrales Delgado [12]. They represented mutations by test purposes which specify the form of test cases that cover corresponding mutations.

Combinations of mutation testing and random testing have also been studied in previous work of Aichernig's group at Graz University of Technology [21, 22]. An important insight gained in this work is that random testing is able to cover a large number of mutations fast. Because of that, only a few subtle mutants need to be checked with directed search techniques. While our learning-focused approach does not aim at covering all mutations, as it does not involve a directed search, mutation coverage provides a certain level of confidence in learning as well. We can guarantee that covered mutations do not affect the learned model.

Furthermore, model-based mutation testing has been applied in the industry [21], it has been applied for real-time [20] and hybrid systems [19], and it is supported by tools, such as MOMUT::UML[1] [169].

Testing Real-Time Systems. Early work on black-box testing of dense real-time systems has been performed by Springintveld et al. [256], who proposed a test-suite generation technique similar to the W-method [83, 277]. To handle dense real-time, they reduced timed transition systems to *grid automata* which are discrete finite automata. While their work is mostly theoretical, because the generated test suites are very large, they showed that complete test suites exist and they formalised assumptions on real-time SUTs that enable testing. In fact, we use these assumptions, but we follow a more practical approach

[1]MOMUT::UML is available online at: `https://momut.org/`, accessed on November 4, 2019.

to testing in Chapter 9. Nielson and Skou also performed early work in the area of real-time conformance testing [217]. They presented a test-case generation technique for non-deterministic but determinisable timed automata. The test-case generation applies a coverage criterion based on equivalence classes in a coarse equivalence class partitioning of the model state space.

The exist variants and extensions of the real-time model-checker UPPAAL [178] for testing real-time systems based on timed automata. UPPAAL COVER generates test cases based on coverage criteria expressed via observer automata [136]. UPPAAL TRON performs online black-box conformance testing with respect to relativized input/output conformance [138]. Our approach to testing in Chapter 9 is similar to that of UPPAAL TRON which also performs random choices. However, we generate test cases offline. Moreover, UPPAAL TRON can handle more general timed automata by solely concentrating on testing, while we focus on model learning. UPPAAL YGGDRASIL is a more recent addition to the UPPAAL tool family, which is fully integrated into the main tool of UPPAAL [162]. This tool generates test cases offline from timed automata to achieve edge coverage and coverage of user-defined requirements expressed as test purposes.

UPPAAL ECDAR [91] was originally developed for compositional design and verification of real-time systems. Larsen et al. [181] have shown how to use it for efficient model-based mutation testing and it was extended to support mutation-based test-case generation natively [133]. Related to that, Lorber et al. [193] demonstrated the application of model checking via UPPAAL to perform requirements-based model-based mutation testing. Model-based mutation testing of dense real-time systems was first proposed by Aichernig et al. [20]. They generated test cases by checking language inclusion between a specification model and its mutated versions. Counterexamples to language inclusion served as test cases, as they show non-conformance between specification and mutant with respect to timed input-output conformance [170]. Motivated by the application for model-based mutation testing, Krenn et al. [168] proposed an incremental procedure for checking language inclusion between networks of deterministic timed automata.

In addition to defining timed input-output conformance [170], an adaptation of **ioco** [270], Krichen and Tripakis presented various test-case generation algorithms for conformance testing of real-time systems, including online and offline algorithms and coverage-guided algorithms. Another adaptation of Tretmans' **ioco** theory [270] to testing of real-time systems has been presented by Brandán Briones and Brinksma [71]. The authors introduced an operational interpretation of quiescent behaviour to define their variant of timed input-output conformance and they also presented a test-case generation algorithm. Basically, they consider a system to be quiescent if it does not interact with its environment for M or more time units, where M is a parameter. Bohnenkamp and Belinfante handled quiescent behaviour similarly in their adaptation of the conformance-testing tool TORX to online testing of real-time systems [64].

12 Conclusion and Outlook

In this thesis, we have discussed learning-based testing and test-based learning with a focus on networked environments, such as the IoT. In exploratory research, we performed a case study in which we applied learning-based testing to the IoT protocol MQTT. The shortcomings and issues that we identified in this case study served as a starting point for further research.

In this chapter, we will first summarise our motivation for learning-based testing in the IoT, the initial research steps that we carried out, and the extended research tackling issues relevant to the IoT. After that, we will provide conclusions by revisiting research questions that we identified following the exploratory research-phase. Eventually, we will close this chapter and this thesis with an outlook on directions for future work.

12.1 Summary

In the introductory chapter, we identified that the ever-growing complexity of software systems is not new in itself. Indeed, the "software crisis" was first discussed more than fifty years ago [238]. In this discussion, it was pointed out that software engineering as a discipline needs to keep up with the growing complexity to deliver trustworthy software.

Factors that have become more relevant in recent years are the increasing connectedness of computer systems and the ubiquity of software-based devices. Especially with the advent of the Internet of Things, these factors are becoming even more dominant. These trends add both to the complexity of software and to the importance of software verification, as devices are nowadays tightly integrated into our everyday lives. We focused on learning-based testing of communication in networked environments for two main reasons. First, reliable communication is essential for the dependability of networked systems. Second, learning-based testing is an approach to verification well-suited for networked environments such as the IoT, since this form of testing is applicable in a black-box setting. It does not require prior information about the internals of the considered systems, which is rarely available, given the large numbers of IoT devices.

12.1.1 Exploratory Research

As a first step towards effective learning-based testing in the IoT, we performed exploratory research in two lines of work. We surveyed work that has been performed in this area and we performed a case study on learning-based testing in the IoT. By applying existing automata-learning approaches combined with differential model-based testing on learned models, we found various errors in implementations of the communication protocol MQTT. Thus, we have demonstrated that learning-based testing can be effective in the IoT. Despite the successful error detection, we also identified shortcomings of existing learning approaches that limit their applicability in certain settings.

We classified the shortcomings into three groups: *runtime*, *uncertain behaviour*, and *time-dependent behaviour*. First, we observed that the runtime of equivalence-query implementations limits the applicability of automata learning. Traditional conformance testing approaches either do not scale or are not systematic, like purely random testing. Second, most automata-learning methods assume deterministic behaviour, hence they cannot cope with uncertainties. Third, these methods usually cannot model time-dependent behaviour. As a result, we had to ignore certain parts of MQTT. Errors with respect to these parts could not be detected by our initial approach to learning-based testing of MQTT brokers.

12.1.2 Contributions

We proposed several improvements to the state of the art in automata learning to mitigate the identified shortcomings. These improvements range from efficient conformance testing in automata learning over learning-based testing of uncertain behaviour to test-based learning of various types of systems. Note that the proposed test-based learning techniques generally apply learning-based testing by generating test cases from learned hypothesis models, while focusing on improvements with respect to learning.

Efficient Testing in Automata Learning. The first shortcoming of automata learning that we tackled was the high runtime required by conformance testing performed during equivalence queries; see Chapter 4. Motivated by previous research [22], we decided to combine fault-based test-case generation with random testing. We proposed a three-step testing process for efficient testing in active automata learning. The first step generates a large number of test cases through random walks on hypothesis automata models. The second step selects a subset of these test cases to optimise the coverage of a certain class of faults. In the third step, these test cases are executed to discover discrepancies between hypotheses and the SUL.

The fault class that shall be covered is governed by so-called mutation operators. We designed operators specifically for active automata learning in the MAT framework. These operators inject faults into hypothesis models to create mutated models that mimic potential successor hypotheses. The rationale behind this form of test-case generation is as follows. By covering a large number of mutated models, test cases are able to cover a large number of alternative SUL behaviours that are consistent with available information about the SUL. Variability introduced by random walks during test-case generation helps to explore the SUL state space more thoroughly.

We evaluated the proposed fault-based testing technique by comparing it to traditional conformance testing techniques and to a state-of-the-art conformance testing technique, which has been first applied in automata learning by Smeenk et al. [254]. The evaluation showed that our technique generally performs well and that it outperforms deterministic conformance testing via the partial W-method [117]. Felix Wallner's bachelor's thesis [288] extended this evaluation by taking different learning algorithms and additional testing techniques into account. Mutation-based testing showed favourable performance in this extended evaluation as well, but it needs to be combined with either the RS learning algorithm [240] or the TTT algorithm [151]. The advanced counterexample processing performed by these two algorithms is crucial for efficient learning.

Learning-Based Testing of Uncertain Behaviour. In order to address the second shortcoming, we studied learning approaches that are able to capture uncertain behaviour and decided to learn stochastic models which model uncertainties probabilistically. To enable learning-based testing of stochastic systems, we proposed probabilistic black-box reachability checking; see Chapter 6. This approach basically generates online-testing strategies to maximise the probability of reaching specified outputs.

For that, we follow an iterative process that starts with random sampling of system traces. After that, the process interleaves learning from samples, generating testing strategies, and property-directed sampling of new traces through testing. The evaluation of the proposed approach showed that it can reliably generate near-optimal testing strategies for stochastic systems in a black-box setting.

Test-Based Learning. We presented various approaches to test-based learning. These approaches focus on the learning aspect, while applying learning-based testing techniques.

In Chapter 7, we adapted the L^* algorithm by Angluin [37] to stochastic systems. More concretely, we presented a novel active learning algorithm that is able to learn stochastic system models from system traces generated via testing. The testing technique applied in equivalence queries is inspired by the coverage-guided testing that we proposed in Chapter 4. Our L^*-based algorithm is able to learn significantly more accurate MDPs than IoAlergia, a state-of-the-art passive learning algorithm, given the same amount of data.

For learning of timed system models, we followed a metaheuristic approach that we discussed in Chapter 8. With that we present a solution to the third shortcoming that we identified. We adopted genetic programming to automatically learn timed automata models from timed traces recorded during testing. In particular, our genetic programming implementation involves mutation and crossover for timed automata, a fine-grained fitness evaluation, and speciation, which splits the metaheuristic search into a local and a global search. The evaluation of the presented technique showed that medium-sized timed automata models can be learned successfully via genetic programming.

The passive genetic-programming-based learning approach for timed systems was extended to an active approach by Andrea Pferscher in her master's thesis [234]; see Chapter 9. She demonstrated that the number of traces required for learning can be reduced significantly through active testing during learning. The applied testing technique is based on random walks, similar to our fault-based testing technique.

Chapter 10 considers test-based learning of hybrid systems, an extension of timed systems. In said chapter, we describe a two-phase process that we implemented. The first phase of the process collects system traces through learning-based testing. The second phase then generalises from these traces by training a recurrent neural network. In order to be able to apply learning-based testing to hybrid systems and to target specific behaviours of interest, such as collisions in platooning control systems, we devised (1) a testing process for abstract test cases and a (2) reachability-property-directed testing technique. The latter is inspired by the testing technique applied in probabilistic black-box reachability checking that we presented in Chapter 6. Neural networks trained on traces sampled through directed testing outperformed neural networks trained on random system traces.

12.2 Conclusions

At the end of every chapter, we discussed our findings with respect to research questions that we identified following our exploratory research. In this section, we revisit all research questions and provide concluding remarks at the end of the section.

RQ 1.1 Are randomised testing techniques a sensible choice for learning-based testing? We developed and applied randomised testing techniques in this thesis to successfully learn various types of systems. In contrast to that, we found that deterministic conformance testing does not scale and generally performs worse than randomised testing techniques. It should be noted, though, that purely random testing fails for large systems. Testing should aim to benefit from intermediate hypothesis models, as our mutation-based testing technique does. The random Wp-method, for instance, also takes hypotheses into account.

In Section 4.5, we showed that mutation-based testing and the random Wp-method outperform deterministic testing via the Wp-method. Moreover, both techniques learn correct models more reliably than purely random testing.

Systems involving uncertain behaviour introduce randomisation into testing through their stochastic nature. Hence, randomised online-testing techniques are generally a sensible choice in this context. The tool JTORX [51, 52], for instance, also applies an online-testing strategy with random choices for systems with uncertain behaviour.

RQ 1.2 What guarantees can be given if randomised testing is applied? Before discussing the guarantees that we can give, it should be noted that it is in general hard or impossible to provide guarantees on the results of automata learning, especially prior to learning. In deterministic test-based learning, it is, for instance, impossible to guarantee that the correct automaton has been learned, unless an upper bound on the SUL states is known. In this case, the W-method [83, 277] could be used to generate an exhaustive test suite. From a practical point of view, exhaustive testing may not be feasible even if an upper bound is known. PAC learnability guarantees are an alternative, but these guarantees usually hold relative to an arbitrary but fixed sampling distribution of system traces [95]. In practice, these guarantees may be useful in cases, where we know a distribution of traces that occurs during system usage. Without this kind of knowledge, PAC guarantees might be misleading.

Despite the apparent difficulty, we are able to give various types of guarantees on learning results and on learning in general.

In Chapter 3, we applied random testing to learn MQTT models. We could not prove that we learned the correct models, but we checked every potential error for spuriousness. Due to the spuriousness check, we can guarantee that every error that we detected is indeed an error.

Deterministic active automata learning, such as L^* [37], comes with a built-in minimality guarantee. This means that learned automata are minimal in the number of states. Hence, if we learn an automaton with 30 states, then this automaton is either equivalent to the SUL or the SUL has strictly more than 30 states. This also holds when randomised testing is applied in equivalence queries. Moreover, more efficient testing generally enables learning of larger automata. Hence, our efficient fault-based conformance testing approach is often able to provide stronger guarantees than deterministic conformance testing.

We learned testing strategies for reachability properties of stochastic systems in Chapter 6. The evaluation step in the process shown in Figure 6.1 computes an estimate of the probability of reaching a specified property with the SUL controlled by a learned strategy. The probability estimate is approximately correct with a specified confidence and error bound. Since we evaluate testing strategies on the actual SUL, the estimate is an approximate lower bound for the true optimal reachability probability.

We have shown that L^*_{MDP}, our active learning technique for MDPs, converges to the true model of the SUL in the limit. This is a weaker result than PAC learnability, though. PAC learnability, however, is hard to achieve for MDPs [199].

RQ 1.3 Can learning with randomised conformance testing reliably generate correct system models? We evaluated active automata learning combined with randomised testing in learning experiments discussed in Section 4.4 and in Section 4.5. These experiments demonstrated that randomised testing technique can reliably learn correct Mealy-machine models. Our fault-based technique and the random Wp-method performed well most consistently in Section 4.5, with the fault-based technique performing best overall.

Randomised testing showed good performance for learning other types of models as well. L^*_{MDP} applies randomised testing and learns accurate models of various types of stochastic systems; see Section 7.6. We were also able to learn timed automata from random timed traces. A manual inspection of learned models revealed that we generally learned the correct model. However, purely random testing may not be sufficient. Learned models should be taken into account during testing. For instance, we demonstrated that using learned models is crucial in the context of learning neural network models of hybrid systems in Chapter 10. Neural networks trained on purely randomly generated test data perform significantly worse than neural networks trained on test data generated via model-based testing. In particular, neural networks trained on data collected through reachability-directed testing achieved a high classification accuracy.

RQ 1.4 Is fault-based testing, such as model-based mutation testing, applicable in automata learning? We demonstrated clearly in Section 4.4 and in Section 4.5 that fault-based testing is applicable in active automata learning and that it performs well. Overall, it performed best among the considered testing techniques. Since our implementation of fault-based testing generates relatively long test cases, it should be combined with learning techniques that apply efficient counterexample processing.

RQ 2.1 Which modelling formalisms are appropriate in learning-based testing of uncertain behaviour? Upon studying the literature on learning models of systems with uncertain behaviour, we decided to capture uncertainties probabilistically in stochastic models. More concretely, we decided to focus on learning Markov decision processes (MDPs). There are three main reasons for that. First, we have seen that implementations of learning algorithms for non-deterministic models assume stochastic system behaviour [161]. This helps, for example, to define a stopping criterion for testing in the implementation of output queries. Hence, it makes sense to directly learn stochastic models. Second, by learning non-deterministic models we do not use all available information. Observed frequencies of outputs are abstracted away in non-deterministic models, whereas they are reflected in transition probabilities of stochastic models. Third, we have chosen MDPs, since they are controllable via inputs, which makes them well-suited for testing.

RQ 2.2 When can we stop learning in the presence of uncertain behaviour? We proposed two different heuristics as stopping criteria in our work on stochastic learning-based testing. In Chapter 6, our goal was to learn testing strategies. Therefore, we developed a stopping criterion which checks if intermediate strategies have likely converged. We basically stop once we observe that several consecutive intermediate testing strategies make similar decisions.

The stopping criterion of L^*_{MDP} is based on the number of (un-)ambiguous traces. Ambiguous traces are traces that could lead to multiple hypothesis states. In other words, we are not certain about the target states of these traces. The stopping criterion applied by L^*_{MDP} aims at decreasing uncertainties in this respect. L^*_{MDP} terminates once the number of ambiguous traces in the learning data structures is low. At this point, the hypothesis structure is likely to be correct.

Since we apply learning algorithms that converge in the limit, we cannot make statements about the quality of models learned from finite samples. Therefore, we rely on heuristics which worked well in practice. Probabilistic black-box reachability with early stopping produced similar results as without early stopping. Models learned with L^*_{MDP} were generally accurate.

RQ 2.3 Is test-based active learning feasible in the presence of stochastic behaviour? Through the development, implementation, and evaluation of L^*_{MDP}, we have demonstrated that test-based active learning of MDPs is feasible and can produce more accurate models than passive learning, given the same amount of data.

RQ 3.1 Is learning of real-time system models feasible through metaheuristic search-based techniques? We successfully learned timed automata from timed traces through genetic programming, a metaheuristic search-based approach. The learned models are sufficiently large to model real-world systems, such as car alarm systems and devices counting particles in exhaust gas. Andrea Pferscher demonstrated in her master's thesis that active testing during learning can increase learning efficiency. Given a limited testing budget, active testing can make learning more reliable; see Chapter 9 [234].

RQ 3.2 What assumptions are sufficient to enable learning in the context of real-time systems? We assume systems to be deterministic, output urgent, and to have isolated outputs. Since we consider a test-based setting, we also assume systems to be input enabled. These assumptions allow us to learn timed automata from positive data that is given in the form of system traces observed during testing.

More concretely, under these assumptions we can view timed automata as functions from test sequences to timed traces [256]. Hence, given a timed trace t observed during testing, we know that every timed trace which is not equivalent to t, but which has the same underlying test sequence, cannot be observed. With these assumptions, we implicitly get negative data from observed positive data. This makes learning feasible. It is known that only very restricted classes of languages can be learned from solely positive data without additional information [95, 122].

In the evaluation of active genetic programming, we found an additional assumption required by the passive approach; see Section 9.3. It requires an approximately correct estimation of c_{max}, the largest constant appearing in clock guards. Active genetic programming can handle inaccurate estimations of c_{max} better by basically learning relevant constants.

Concluding Remarks. Finally, we want to discuss how the work presented within this thesis and the aforementioned findings specifically relate to the thesis statement. The thesis statement reads as follows:

> Test-case generation based on intermediate learned automata models combined with randomisation enables active learning of accurate automata models.

In this thesis, we presented various test-case generation techniques that derive test cases from intermediate learned models including fault-based, online reachability-directed, transition-coverage-based, and random-walk-based techniques. As discussed above, we were able to reliably learn accurate models of deterministic, stochastic, and timed systems. Hence, the thesis statement holds. Testing based on learned models enables learning of accurate automata models of black-box systems. In addition, we also successfully learned accurate neural network models of hybrid systems by applying learning-based testing.

12.3 Future Work

We contributed in various ways to the state of the art in automata learning. Our contributions include algorithms, testing techniques, approaches to learning, benchmark models, and prototypical implementations. In addition to that, we identified various directions for future research to extend and improve our work. The following paragraphs provide an overview of potential future work.

Extending Fault-Based Testing. We presented fault-based testing for Mealy-machine models. As pointed out in Section 1.6.4, these models may be too restrictive in certain scenarios. We concentrated mostly on extending learning to stochastic and timed systems. Another potential extension is the inclusion of data variables into learned models. There exist active learning approaches for that, but specialised conformance-testing techniques for these models rarely exist. Cassel et al. [76], for instance, learn extended finite-state machines and they note that equivalence checking could be implemented via conformance testing. We believe that fault-based testing would be very well-suited in this context. Our split-state mutation operators are specialised towards active automata learning and additional mutation operators could target constraints on data variables. In fact, Bernhard Aichernig's group worked intensively on efficient fault-based test-case generation from models including data variables [14, 23, 169].

Probabilistic Black-Box Checking. We learned testing strategies for stochastic systems with respect to reachability properties. A natural direction for future research would be to consider more general properties, for instance, given in PCTL. This may require more general schedulers, that is, memoryless schedulers may not be sufficient. Another potential extension of this work would be to maximise rewards or to minimise costs, such as power consumption, rather than maximising a reachability probability.

Probabilistic black-box reachability checking could also be integrated into L^*_{MDP} by performing reachability-property-directed sampling as part of equivalence queries. Since L^*_{MDP} produced more accurate models than IoAlergia, the combination of L^*_{MDP} and reachability checking may perform better than the approach discussed in Chapter 6.

PAC Learnability of MDPs. We showed that L^*_{MDP} converges to the correct model in the large sample limit. This means that L^*_{MDP} eventually learns an MDP which is isomorphic to the canonical MDP underlying the SUL. However, we did not prove PAC learning results. PAC learning would allow to make statements about the required sample size to achieve some desired model accuracy. Hence, practical applications might benefit from this kind of guarantees. An obstacle to overcome towards PAC learning is the identification of a distance measure that is adequate in a verification context [199]. It should, for instance, be possible to relate distance values to the accuracy of model checking results.

Learning Stochastic Time-Dependent Behaviour. This thesis presented learning approaches for discrete-time stochastic systems and for deterministic real-time systems. A natural next step would be to tackle the problem of learning models of stochastic real-time systems, such as stochastic timed automata, which can be analysed with Uppaal SMC [94]. There are various possible ways to approach this problem. We could extend L^*_{MDP} to consider time as well and integrate ideas from other existing approaches [96, 199, 280]. Alternatively, we could adapt the genetic programming to learn stochastic timed automata by computing fitness based on some distance measure.

Meta-Optimisation of Parameters. The techniques presented within this thesis are controlled by parameterised probabilistic choices. In some cases, such as the genetic programming of timed automata, the parameter space is relatively large and concrete configurations heavily affect performance. We generally strived to provide guidelines for choosing parameters and to identify parameter settings that worked well across all experiments. Meta-optimisation of parameters, which is an automatic search for adequate parameter values, might ease the application of our work in new environments.

Monitoring of Hybrid Systems. We trained neural networks to detect crashes in a platooning system as soon as they happen. Hence, we could run the platooning system and a neural network in parallel and the neural network would output *crash* once the platooning system crashes. This is not very useful for monitoring, since crash detection exactly at the time of crashing is obviously too late. However, note that we have chosen crash detection as an illustrative example. In future work, we plan to extend this work to the detection of dangerous situations that might lead to crashes. This would be more useful for monitoring, provided that there is sufficient time to intervene.

Closing Remark. In conclusion, we believe that this thesis constitutes a valuable contribution to the field of automata learning and in particular to automata learning-based testing. We hope that the presented techniques and ideas prove useful and provide inspiration to other researchers.

Bibliography

[1] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010. ISBN 978-3-642-16572-6. doi: 10.1007/978-3-642-16573-3_14. URL `https://doi.org/10.1007/978-3-642-16573-3_14`. (Cited on pages 6 and 30.)

[2] Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and abstraction of the biometric passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010. ISBN 978-3-642-16557-3. doi: 10.1007/978-3-642-16558-0_54. URL `https://doi.org/10.1007/978-3-642-16558-0_54`. (Cited on page 204.)

[3] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, 2012. ISBN 978-3-642-32758-2. doi: 10.1007/978-3-642-32759-9_4. URL `https://doi.org/10.1007/978-3-642-32759-9_4`. (Cited on pages 31, 37, 39 and 207.)

[4] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W. Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In Jeffrey Heinz, Colin de la Higuera, and Tim Oates, editors, *Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012, University of Maryland, College Park, USA, September 5-8, 2012*, volume 21 of *JMLR Proceedings*, pages 4–18. JMLR.org, 2012. URL `http://proceedings.mlr.press/v21/aarts12a.html`. (Cited on pages 33, 204, 206 and 207.)

[5] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W. Vaandrager, and Sicco Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96 (1-2):189–224, 2014. doi: 10.1007/s10994-013-5405-0. URL `https://doi.org/10.1007/s10994-013-5405-0`. (Cited on pages 17, 33, 204, 206 and 207.)

[6] Fides Aarts, Paul Fiterău-Broştean, Harco Kuppens, and Frits W. Vaandrager. Learning register automata with fresh value generation. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015. ISBN 978-3-319-25149-3. doi: 10.1007/978-3-319-25150-9_11. URL `https://doi.org/10.1007/978-3-319-25150-9_11`. (Cited on pages 36 and 39.)

[7] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015. doi: 10.1007/s10703-014-0216-x. URL `https://doi.org/10.1007/s10703-014-0216-x`. (Cited on pages 31 and 37.)

[8] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1016–1026. ACM, 2018. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180160. URL `https://doi.org/10.1145/3180155.3180160`. (Cited on page 212.)

[9] Gusztáv Adamis, Gábor Kovács, and György Réthy. Generating performance test model from conformance test logs. In Joachim Fischer, Markus Scheidgen, Ina Schieferdecker, and Rick Reed, editors, *SDL 2015: Model-Driven Engineering for Smart Cities - 17th International SDL Forum, Berlin, Germany, October 12-14, 2015, Proceedings*, volume 9369 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2015. ISBN 978-3-319-24911-7. doi: 10.1007/978-3-319-24912-4_19. URL `https://doi.org/10.1007/978-3-319-24912-4_19`. (Cited on page 207.)

[10] Bernhard Aichernig, Roderick Bloem, Franz Pernkopf, Franz Röck, Tobias Schrank, and Martin Tappler. Poster: Learning models of a network protocol using neural network language models. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016. ISBN 978-1-5090-0824-7. URL `https://www.ieee-security.org/TC/SP2016/poster-abstracts/36-poster_abstract.pdf`. (Cited on page 12.)

[11] Bernhard K. Aichernig. Model-based mutation testing of reactive systems – from semantics to automated test-case generation. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2013. ISBN 978-3-642-39697-7. doi: 10.1007/978-3-642-39698-4_2. URL `https://doi.org/10.1007/978-3-642-39698-4_2`. (Cited on pages 7 and 213.)

[12] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2006. ISBN 3-540-33093-3. doi: 10.1007/11693017_24. URL `https://doi.org/10.1007/11693017_24`. (Cited on pages 41 and 213.)

[13] Bernhard K. Aichernig and Richard Schumi. How fast is MQTT? - statistical model checking and testing of IoT protocols. In Annabelle McIver and Andras Horvath, editors, *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*, volume 11024 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2018. ISBN 978-3-319-99153-5. doi: 10.1007/978-3-319-99154-2_3. URL `https://doi.org/10.1007/978-3-319-99154-2_3`. (Cited on pages 45, 207 and 211.)

[14] Bernhard K. Aichernig and Martin Tappler. Symbolic input-output conformance checking for model-based mutation testing. *Electronic Notes in Theoretical Computer Science*, 320:3–19, 2016. doi: 10.1016/j.entcs.2016.01.002. URL `https://doi.org/10.1016/j.entcs.2016.01.002`. (Cited on pages 41 and 220.)

[15] Bernhard K. Aichernig and Martin Tappler. Learning from faults: Mutation testing in active automata learning. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal*

Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings, volume 10227 of *Lecture Notes in Computer Science*, pages 19–34, 2017. ISBN 978-3-319-57287-1. doi: 10.1007/978-3-319-57288-8_2. URL `https://doi.org/10.1007/978-3-319-57288-8_2`. (Cited on pages 9, 10, 11, 12, 23, 29, 55, 114, 203 and 252.)

[16] Bernhard K. Aichernig and Martin Tappler. Probabilistic black-box reachability checking. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2017. ISBN 978-3-319-67530-5. doi: 10.1007/978-3-319-67531-2_4. URL `https://doi.org/10.1007/978-3-319-67531-2_4`. (Cited on pages 9, 11, 91, 92, 99, 203 and 252.)

[17] Bernhard K. Aichernig and Martin Tappler. Efficient active automata learning via mutation testing. *Journal of Automated Reasoning*, 63(4):1103–1134, 2019. doi: 10.1007/s10817-018-9486-0. URL `https://doi.org/10.1007/s10817-018-9486-0`. (Cited on pages 9, 11, 12, 15, 23, 29, 55, 65, 68, 114, 144, 171, 195, 196, 203 and 252.)

[18] Bernhard K. Aichernig and Martin Tappler. Probabilistic black-box reachability checking (extended version). *Formal Methods in System Design*, 54(3):416–448, May 2019. ISSN 1572-8102. doi: 10.1007/s10703-019-00333-0. URL `https://doi.org/10.1007/s10703-019-00333-0`. (Cited on pages 4, 9, 11, 91, 92, 94, 99, 203 and 252.)

[19] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2009. ISBN 978-3-642-17070-6. doi: 10.1007/978-3-642-17071-3_12. URL `https://doi.org/10.1007/978-3-642-17071-3_12`. (Cited on pages 7 and 213.)

[20] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013. ISBN 978-3-642-38915-3. doi: 10.1007/978-3-642-38916-0_2. URL `https://doi.org/10.1007/978-3-642-38916-0_2`. (Cited on pages 7, 174, 213 and 214.)

[21] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014. ISBN 978-3-319-09098-6. doi: 10.1007/978-3-319-09099-3_1. URL `https://doi.org/10.1007/978-3-319-09099-3_1`. (Cited on pages 7, 58, 174 and 213.)

[22] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification & Reliability*, 25(8):716–748, 2015. doi: 10.1002/stvr.1522. URL `https://doi.org/10.1002/stvr.1522`. (Cited on pages 7, 16, 53, 57, 58, 60, 63, 213 and 216.)

[23] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97:383–404, 2015. doi: 10.1016/j.scico.2014.05.004. URL `https://doi.org/10.1016/j.scico.2014.05.004`. (Cited on pages 7, 16, 41, 213 and 220.)

[24] Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Tappler, and Johannes Winter. Automata learning for symbolic execution. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. ISBN 978-0-9835678-8-2. doi: 10.23919/FMCAD.2018.8602991. URL `https://doi.org/10.23919/FMCAD.2018.8602991`. (Cited on page 12.)

[25] Bernhard K. Aichernig, Severin Kann, and Richard Schumi. Statistical model checking of response times for different system deployments. In Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings*, volume 10998 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2018. ISBN 978-3-319-99932-6. doi: 10.1007/978-3-319-99933-3_11. URL `https://doi.org/10.1007/978-3-319-99933-3_11`. (Cited on page 211.)

[26] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 74–100. Springer, 2018. ISBN 978-3-319-96561-1. doi: 10.1007/978-3-319-96562-8_3. URL `https://doi.org/10.1007/978-3-319-96562-8_3`. (Cited on pages 3, 4, 6, 9, 10, 24, 29, 56, 203 and 204.)

[27] Bernhard K. Aichernig, Priska Bauerstätter, Elisabeth Jöbstl, Severin Kann, Robert Korosec, Willibald Krenn, Cristinel Mateis, Rupert Schlick, and Richard Schumi. Learning and statistical model checking of system response times. *Software Quality Journal*, 27(2): 757–795, 2019. doi: 10.1007/s11219-018-9432-8. URL `https://doi.org/10.1007/s11219-018-9432-8`. (Cited on page 211.)

[28] Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Horn, Franz Pernkopf, Wolfgang Roth, Astrid Rupp, Martin Tappler, and Markus Tranninger. Learning a behavior model of hybrid systems through combining model-based testing and machine learning (full version). *CoRR*, abs/1907.04708, 2019. URL `http://arxiv.org/abs/1907.04708`. (Cited on pages 12, 197 and 203.)

[29] Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Horn, Franz Pernkopf, Wolfgang Roth, Astrid Rupp, Martin Tappler, and Markus Tranninger. Learning a behavior model of hybrid systems through combining model-based testing and machine learning. In Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall, editors, *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*, volume 11812 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019. ISBN 978-3-030-31279-4. doi: 10.1007/978-3-030-31280-0_1. URL `https://doi.org/10.1007/978-3-030-31280-0_1`. **Best Paper at ICTSS 2019**. (Cited on pages 9, 11, 189, 203 and 212.)

[30] Bernhard K. Aichernig, Franz Pernkopf, Richard Schumi, and Andreas Wurm. Predicting and testing latencies with deep learning: An IoT case study. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs - 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, volume 11823 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2019. ISBN 978-3-030-31156-8. doi: 10.1007/978-3-030-31157-5_7. URL `https://doi.org/10.1007/978-3-030-31157-5_7`. (Cited on pages 207 and 211.)

[31] Rajeev Alur. Can we verify cyber-physical systems?: Technical perspective. *Communications of the ACM*, 56(10):96, 2013. doi: 10.1145/2507771.2507782. URL `https://doi.org/10.1145/2507771.2507782`. (Cited on page 189.)

[32] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 (2):183–235, 1994. doi: 10.1016/0304-3975(94)90010-8. URL `https://doi.org/10.1016/0304-3975(94)90010-8`. (Cited on pages 38, 161 and 163.)

[33] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems. HS 1992, HS 1991*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992. ISBN 3-540-57318-6. doi: 10.1007/3-540-57318-6_30. URL `https://doi.org/10.1007/3-540-57318-6_30`. (Cited on page 189.)

[34] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273, 1999. doi: 10.1016/S0304-3975(97)00173-4. URL `https://doi.org/10.1016/S0304-3975(97)00173-4`. (Cited on pages 162 and 210.)

[35] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109. ACM, 2005. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040314. URL `https://doi.org/10.1145/1040305.1040314`. (Cited on page 205.)

[36] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 978-0-521-88038-1. doi: 10.1017/CBO9780511809163. URL `https://doi.org/10.1017/CBO9780511809163`. (Cited on page 2.)

[37] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. doi: 10.1016/0890-5401(87)90052-6. URL `https://doi.org/10.1016/0890-5401(87)90052-6`. (Cited on pages 4, 7, 10, 15, 17, 18, 20, 21, 22, 29, 62, 68, 69, 77, 79, 91, 92, 99, 100, 103, 129, 131, 132, 133, 161, 171, 195, 203, 205, 208, 209, 211, 217 and 218.)

[38] Hugo L. S. Araujo, Gustavo Carvalho, Morteza Mohaqeqi, Mohammad Reza Mousavi, and Augusto Sampaio. Sound conformance testing for cyber-physical systems: Theory and implementation. *Science of Computer Programming*, 162:35–54, 2018. doi: 10.1016/j.scico.2017.07.002. URL `https://doi.org/10.1016/j.scico.2017.07.002`. (Cited on page 189.)

[39] 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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1690–1701. ACM, 2016. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978383. URL `http://doi.acm.org/10.1145/2976749.2978383`. (Cited on page 204.)

[40] Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with quviq QuickCheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. ISBN 1-59593-490-1. doi: 10.1145/1159789.1159792. URL `https://doi.org/10.1145/1159789.1159792`. (Cited on page 206.)

[41] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447 – 452, 1998. ISSN 1474-6670. doi: https://doi.org/10.1016/S1474-6670(17)42032-5. URL `http://www.sciencedirect.com/science/article/pii/S1474667017420325`. Special issue on the 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July. (Cited on page 182.)

[42] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi: 10.1016/0196-6774(90)90021-6. URL `https://doi.org/10.1016/0196-6774(90)90021-6`. (Cited on pages 109, 116 and 155.)

[43] Giorgio Bacci, Giovanni Bacci, Kim Guldstrand Larsen, and Radu Mardare. MDPDist library. `http://people.cs.aau.dk/~giovbacci/tools/bisimdist.zip`, accessed on November 4, 2019. (Cited on page 153.)

[44] Giorgio Bacci, Giovanni Bacci, Kim Guldstrand Larsen, and Radu Mardare. Computing behavioral distances, compositionally. In Krishnendu Chatterjee and Jirí Sgall, editors, *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, volume 8087 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 2013. ISBN 978-3-642-40312-5. doi: 10.1007/978-3-642-40313-2_9. URL `https://doi.org/10.1007/978-3-642-40313-2_9`. (Cited on pages 126 and 152.)

[45] Giorgio Bacci, Giovanni Bacci, Kim Guldstrand Larsen, and Radu Mardare. The BisimDist library: Efficient computation of bisimilarity distances for Markovian models. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 278–281. Springer, 2013. ISBN 978-3-642-40195-4. doi: 10.1007/978-3-642-40196-1_23. URL `https://doi.org/10.1007/978-3-642-40196-1_23`. (Cited on page 152.)

[46] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008. ISBN 978-0-262-02649-9. (Cited on pages 5, 92, 94, 96 and 121.)

[47] Rena Bakhshi and Ansgar Fehnker. On the impact of modelling choices for distributed information spread. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 41–50. IEEE Computer Society, 2009. ISBN 978-0-7695-3808-2. doi: 10.1109/QEST.2009.37. URL `https://doi.org/10.1109/QEST.2009.37`. (Cited on page 98.)

[48] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios E. Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 135–175. Springer, 2018. ISBN 978-3-319-75631-8. doi: 10.1007/978-3-319-75632-5_5. URL `https://doi.org/10.1007/978-3-319-75632-5_5`. (Cited on page 189.)

[49] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. ISBN 3-540-23068-8. doi: 10.1007/978-3-540-30080-9_7. URL `https://doi.org/10.1007/978-3-540-30080-9_7`. (Cited on page 174.)

[50] Boris Beizer. *Software Testing Techniques (2nd ed.).* Van Nostrand Reinhold, 1990. ISBN 978-0-442-20672-7. (Cited on page 2.)

[51] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010. ISBN 978-3-642-12001-5. doi: 10.1007/978-3-642-12002-2_21. URL `https://doi.org/10.1007/978-3-642-12002-2_21`. (Cited on pages 122, 213 and 218.)

[52] Axel Belinfante. *JTorX: Exploring Model-Based Testing*. PhD thesis, University of Twente, Netherlands, 2014. IPA Dissertation series no. 2014-09. (Cited on pages 122 and 218.)

[53] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems, September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999. ISBN 0-7923-8581-0. (Cited on page 213.)

[54] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005. ISBN 3-540-25420-X. doi: 10.1007/978-3-540-31984-9_14. URL `https://doi.org/10.1007/978-3-540-31984-9_14`. (Cited on pages 4, 29, 30, 33, 56 and 87.)

[55] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. *Electronic Notes in Theoretical Computer Science*, 118:3–18, 2005. doi: 10.1016/j.entcs.2004.12.015. URL `https://doi.org/10.1016/j.entcs.2004.12.015`. (Cited on page 208.)

[56] Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM Journal on Computing*, 25(6):1268–1280, 1996. doi: 10.1137/S009753979326091X. URL `https://doi.org/10.1137/S009753979326091X`. (Cited on page 210.)

[57] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 535–552. IEEE Computer Society, 2015. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.39. URL `https://doi.org/10.1109/SP.2015.39`. (Cited on pages 1 and 213.)

[58] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. *Communications of the ACM*, 60(2):99–107, 2017. doi: 10.1145/3023357. URL `https://doi.org/10.1145/3023357`. (Cited on page 213.)

[59] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002. doi: 10.1023/A:1015059928466. URL `https://doi.org/10.1023/A:1015059928466`. (Cited on page 166.)

[60] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972. doi: 10.1109/TC.1972.5009015. URL `https://doi.org/10.1109/TC.1972.5009015`. (Cited on pages 4 and 210.)

[61] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018. doi: 10.1145/3213846.3213872. URL `https://doi.org/10.1145/3213846.3213872`. (Cited on page 205.)

[62] Arianna Blasi, Mauro Pezzè, Alessandra Gorla, and Michael D. Ernst. Research on NLP for RE at università della svizzera italiana (USI): A report. In Paola Spoletini, Patrick Mäder, Daniel M. Berry, Fabiano Dalpiaz, Maya Daneva, Alessio Ferrari, Xavier Franch, Sarah Gregory, Eduard C. Groen, Andrea Herrmann, Anne Hess, Frank Houdek, Oliver Karras, Anne Koziolek, Kim Lauenroth, Cristina Palomares, Mehrdad Sabetzadeh, Norbert Seyff, Marcus Trapp, Andreas Vogelsang, and Thorsten Weyer, editors, *Joint Proceedings of REFSQ-2019 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 25th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2019), Essen, Germany, March 18th, 2019*, volume 2376 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL `http://ceur-ws.org/Vol-2376/NLP4RE19_paper17.pdf`. (Cited on page 205.)

[63] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996. doi: 10.1162/evco.1996.4.4.361. URL `https://doi.org/10.1162/evco.1996.4.4.361`. (Cited on page 170.)

[64] Henrik C. Bohnenkamp and Axel Belinfante. Timed testing with TorX. In John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005. ISBN 3-540-27882-6. doi: 10.1007/11526841_13. URL `https://doi.org/10.1007/11526841_13`. (Cited on pages 162 and 214.)

[65] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 1997. ISBN 3-540-65493-3. doi: 10.1007/3-540-49213-5_5. URL `https://doi.org/10.1007/3-540-49213-5_5`. (Cited on page 164.)

[66] Matko Botincan and Vedran Novakovic. Model-based testing of the conference protocol with Spec Explorer. In *9th International Conference on Telecommunications – ConTEL 2007, June 13-15, 2007, Zagreb, Croatia*, pages 131–138, 2007. doi: 10.1109/CONTEL.2007.381861. URL `https://dx.doi.org/10.1109/CONTEL.2007.381861`. (Cited on page 213.)

[67] Redouane Bouchekir and Mohand Cherif Boukala. Learning-based symbolic assume-guarantee reasoning for Markov decision process by using interval Markov process. *Innovations in Systems and Software Engineering*, 14(3):229–244, Sep 2018. ISSN 1614-5054. doi: 10.1007/s11334-018-0316-7. URL `https://doi.org/10.1007/s11334-018-0316-7`. (Cited on page 210.)

[68] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. Automated conformance verification of hybrid systems. In Ji Wang, W. K. Chan, and Fei-Ching Kuo, editors, *Proceedings*

of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010, pages 3–12. IEEE Computer Society, 2010. doi: 10.1109/QSIC.2010.53. URL `https://doi.org/10.1109/QSIC.2010.53`. (Cited on page 41.)

[69] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. Verification of Markov decision processes using learning algorithms. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2014. ISBN 978-3-319-11935-9. doi: 10.1007/978-3-319-11936-6_8. URL `http://dx.doi.org/10.1007/978-3-319-11936-6_8`. (Cited on pages 95, 104 and 211.)

[70] Nicolas Brémond and Roland Groz. Case studies in learning models and testing without reset. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, pages 40–45. IEEE, 2019. ISBN 978-1-7281-0888-9. doi: 10.1109/ICSTW.2019.00030. URL `https://doi.org/10.1109/ICSTW.2019.00030`. (Cited on page 208.)

[71] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004. ISBN 3-540-25109-X. doi: 10.1007/978-3-540-31848-4_5. URL `https://doi.org/10.1007/978-3-540-31848-4_5`. (Cited on page 214.)

[72] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985. doi: 10.1016/0096-0551(85)90011-6. URL `https://doi.org/10.1016/0096-0551(85)90011-6`. (Cited on page 213.)

[73] Edited by Andrew Banks and Rahul Gupta. MQTT Version 3.1.1. OASIS Standard, October 2014. URL `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`. Latest version: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, accessed on November 4, 2019. (Cited on pages 6, 8, 34, 35, 38, 39, 40, 46, 47, 48, 69 and 113.)

[74] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer, 1994. ISBN 3-540-58473-0. doi: 10.1007/3-540-58473-0_144. URL `https://doi.org/10.1007/3-540-58473-0_144`. (Cited on pages 4, 52, 95, 100 and 209.)

[75] Rafael C. Carrasco and José Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO – Theoretical Informatics and Applications (RAIRO: ITA)*, 33 (1):1–20, 1999. doi: 10.1051/ita:1999102. URL `https://doi.org/10.1051/ita:1999102`. (Cited on pages 138, 139, 146, 147, 148 and 209.)

[76] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016. doi: 10.1007/s00165-016-0355-5. URL `https://doi.org/10.1007/s00165-016-0355-5`. (Cited on pages 36 and 220.)

[77] Jorge Castro and Ricard Gavaldà. Learning probability distributions generated by finite-state machines. In Jeffrey Heinz and José M. Sempere, editors, *Topics in Grammatical Inference*, pages 113–142. Springer, Berlin, Heidelberg, 2016. ISBN 978-3-662-48395-4. doi: 10.1007/

978-3-662-48395-4_5. URL https://doi.org/10.1007/978-3-662-48395-4_5. (Cited on pages 52 and 210.)

[78] Taolue Chen, Marta Z. Kwiatkowska, David Parker, and Aistis Simaitis. Verifying team formation protocols with probabilistic model checking. In João Leite, Paolo Torroni, Thomas Ågotnes, Guido Boella, and Leon van der Torre, editors, *Computational Logic in Multi-Agent Systems - 12th International Workshop, CLIMA XII, Barcelona, Spain, July 17-18, 2011. Proceedings*, volume 6814 of *Lecture Notes in Computer Science*, pages 190–207. Springer, 2011. ISBN 978-3-642-22358-7. doi: 10.1007/978-3-642-22359-4_14. URL https://doi.org/10.1007/978-3-642-22359-4_14. (Cited on page 98.)

[79] Yingke Chen and Thomas Dyhre Nielsen. Active learning of Markov decision processes for system verification. In *11th International Conference on Machine Learning and Applications, ICMLA, Boca Raton, FL, USA, December 12-15, 2012. Volume 2*, pages 289–294. IEEE, 2012. doi: 10.1109/ICMLA.2012.158. URL http://dx.doi.org/10.1109/ICMLA.2012.158. (Cited on pages 100, 107 and 209.)

[80] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL http://static.usenix.org/events/sec11/tech/full_papers/Cho.pdf. (Cited on page 205.)

[81] Wontae Choi, George C. Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 623–640. ACM, 2013. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509552. URL https://doi.org/10.1145/2509136.2509552. (Cited on page 207.)

[82] François Chollet et al. Keras, 2015. https://keras.io, accessed on November 4, 2019. (Cited on page 197.)

[83] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978. doi: 10.1109/TSE.1978.231496. URL https://doi.org/10.1109/TSE.1978.231496. (Cited on pages 26, 27, 29, 43, 53, 56, 57, 79, 99, 100, 143, 204, 205, 208, 213 and 218.)

[84] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266. (Cited on pages 37 and 206.)

[85] Edmund M. Clarke and Paolo Zuliani. Statistical model checking for cyber-physical systems. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2011. ISBN 978-3-642-24371-4. doi: 10.1007/978-3-642-24372-1_1. URL https://doi.org/10.1007/978-3-642-24372-1_1. (Cited on page 189.)

[86] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. doi: 10.1007/978-3-319-10575-8. URL https://doi.org/10.1007/978-3-319-10575-8. (Cited on page 5.)

[87] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003. ISBN 3-540-00898-5. doi: 10.1007/3-540-36577-X_24. URL `https://doi.org/10.1007/3-540-36577-X_24`. (Cited on page 4.)

[88] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An interactive learning competition. In Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce W. Watson, editors, *Finite-State Methods and Natural Language Processing, 8th International Workshop, FSMNLP 2009, Pretoria, South Africa, July 21-24, 2009, Revised Selected Papers*, volume 6062 of *Lecture Notes in Computer Science*, pages 139–146. Springer, 2009. ISBN 978-3-642-14683-1. doi: 10.1007/978-3-642-14684-8_15. URL `https://doi.org/10.1007/978-3-642-14684-8_15`. (Cited on pages 29, 57, 58, 69 and 208.)

[89] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 110–125. IEEE Computer Society, 2009. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.14. URL `https://doi.org/10.1109/SP.2009.14`. (Cited on page 207.)

[90] Pedro D'Argenio, Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Smart sampling for lightweight verification of Markov decision processes. *International Journal on Software Tools for Technology Transfer*, 17(4):469–484, 2015. doi: 10.1007/s10009-015-0383-0. URL `https://doi.org/10.1007/s10009-015-0383-0`. (Cited on page 211.)

[91] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2010. ISBN 978-3-642-15642-7. doi: 10.1007/978-3-642-15643-4_29. URL `https://doi.org/10.1007/978-3-642-15643-4_29`. (Cited on page 214.)

[92] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: A complete specification theory for real-time systems. In Karl Henrik Johansson and Wang Yi, editors, *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 91–100. ACM, 2010. ISBN 978-1-60558-955-8. doi: 10.1145/1755952.1755967. URL `https://doi.org/10.1145/1755952.1755967`. (Cited on page 162.)

[93] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Didier Lime, Mathias Grund Sørensen, and Jakob Haahr Taankvist. On time with minimal expected cost! In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2014. ISBN 978-3-319-11935-9. doi: 10.1007/978-3-319-11936-6_10. URL `http://dx.doi.org/10.1007/978-3-319-11936-6_10`. (Cited on page 211.)

[94] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015. doi: 10.1007/s10009-014-0361-y. URL `https://doi.org/10.1007/s10009-014-0361-y`. (Cited on page 221.)

[95] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010. ISBN 0521763169, 9780521763165. (Cited on pages 4, 29, 52, 145, 171, 203, 218 and 220.)

[96] André de Matos Pedro, Paul Andrew Crocker, and Simão Melo de Sousa. Learning stochastic timed automata from sample executions. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 508–523. Springer, 2012. ISBN 978-3-642-34025-3. doi: 10.1007/978-3-642-34026-0_38. URL `https://doi.org/10.1007/978-3-642-34026-0_38`. (Cited on pages 210 and 221.)

[97] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 193–206. USENIX Association, 2015. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter`. (Cited on pages 15, 31, 37, 38, 50, 52, 69, 74, 80, 121, 205, 207 and 208.)

[98] Joeri de Ruiter and Erik Poll. TLS – learned models, 2015. `http://www.cs.ru.nl/J.deRuiter/download/usenix15.zip`, accessed on November 4, 2019. (Cited on pages 69 and 74.)

[99] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136. URL `https://doi.org/10.1109/C-M.1978.218136`. (Cited on page 213.)

[100] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012. doi: 10.1109/JPROC.2011.2160929. URL `https://doi.org/10.1109/JPROC.2011.2160929`. (Cited on page 189.)

[101] Victor S. Dolk, Jeroen Ploeg, and W. P. Maurice H. Heemels. Event-triggered control for string-stable vehicle platooning. *IEEE Trans. Intelligent Transportation Systems*, 18(12):3486–3500, 2017. doi: 10.1109/TITS.2017.2738446. URL `https://doi.org/10.1109/TITS.2017.2738446`. (Cited on page 191.)

[102] Marie Duflot, Laurent Fribourg, Thomas Hérault, Richard Lassaigne, Frédéric Magniette, Stéphane Messika, Sylvain Peyronnet, and Claudine Picaronny. Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC. *Electronic Notes in Theoretical Computer Science*, 128(6):195–214, 2005. doi: 10.1016/j.entcs.2005.04.012. URL `https://doi.org/10.1016/j.entcs.2005.04.012`. (Cited on page 98.)

[103] Edith Elkind, Blaise Genest, Doron A. Peled, and Hongyang Qu. Grey-box checking. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, volume 4229 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2006. ISBN 3-540-46219-8. doi: 10.1007/11888116_30. URL `https://doi.org/10.1007/11888116_30`. (Cited on page 205.)

[104] Sandra Camargo Pinto Ferraz Fabbri, Márcio Eduardo Delamaro, José Carlos Maldonado, and Paulo César Masiero. Mutation analysis testing for finite state machines. In *5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6-9, 1994*, pages 220–229. IEEE, 1994. ISBN 0-8186-6665-X. doi: 10.1109/ISSRE.1994.341378. URL `https://doi.org/10.1109/ISSRE.1994.341378`. (Cited on page 61.)

[105] Ansgar Fehnker and Peng Gao. Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In Thomas Kunz and S. S. Ravi, editors, *Ad-Hoc, Mobile, and Wireless Networks, 5th International Conference, ADHOC-NOW 2006, Ottawa, Canada, August 17-19, 2006, Proceedings*, volume 4104 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 2006. ISBN 3-540-37246-6. doi: 10.1007/11814764_12. URL `https://doi.org/10.1007/11814764_12`. (Cited on page 98.)

[106] Lu Feng, Tingting Han, Marta Z. Kwiatkowska, and David Parker. Learning-based compositional verification for synchronous probabilistic systems. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 511–521. Springer, 2011. ISBN 978-3-642-24371-4. doi: 10.1007/978-3-642-24372-1_40. URL `https://doi.org/10.1007/978-3-642-24372-1_40`. (Cited on page 210.)

[107] Alessandro Fermi, Maurizio Mongelli, Marco Muselli, and Enrico Ferrari. Identification of safety regions in vehicle platooning via machine learning. In *14th IEEE International Workshop on Factory Communication Systems, WFCS 2018, Imperia, Italy, June 13-15, 2018*, pages 1–4. IEEE, 2018. ISBN 978-1-5386-1066-4. doi: 10.1109/WFCS.2018.8402372. URL `https://doi.org/10.1109/WFCS.2018.8402372`. (Cited on page 212.)

[108] Jean-Claude Fernandez and Laurent Mounier. "On the fly" verification of behavioural equivalences and preorders. In Kim Guldstrand Larsen and Arne Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, 1991. ISBN 3-540-55179-4. doi: 10.1007/3-540-55179-4_18. URL `https://doi.org/10.1007/3-540-55179-4_18`. (Cited on pages 41 and 44.)

[109] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997. doi: 10.1016/S0167-6423(96)00032-9. URL `https://doi.org/10.1016/S0167-6423(96)00032-9`. (Cited on page 213.)

[110] Paul Fiterău-Broştean. TCP models, 2016. `https://gitlab.science.ru.nl/pfiteraubrostean/tcp-learner/tree/cav-aec/models`, accessed on November 4, 2019. (Cited on pages 69 and 114.)

[111] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. Learning fragments of the TCP network protocol. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, volume 8718 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2014. ISBN 978-3-319-10701-1. doi: 10.1007/978-3-319-10702-8_6. URL `https://doi.org/10.1007/978-3-319-10702-8_6`. (Cited on page 207.)

[112] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. ISBN 978-3-319-41539-0. doi: 10.1007/978-3-319-41540-6_25. URL `https://doi.org/10.1007/978-3-319-41540-6_25`. (Cited on pages 15, 30, 31, 37, 41, 52, 57, 68, 69, 71, 79, 109, 114, 121, 207 and 208.)

[113] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In Hakan Erdogmus and

Klaus Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 142–151. ACM, 2017. ISBN 978-1-4503-5077-8. doi: 10.1145/3092282.3092289. URL `https://doi.org/10.1145/3092282.3092289`. (Cited on pages 31, 52, 57, 121, 207 and 208.)

[114] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer, 2011. ISBN 978-3-642-21454-7. doi: 10.1007/978-3-642-21455-4_3. URL `http://dx.doi.org/10.1007/978-3-642-21455-4_3`. (Cited on pages 92, 93, 94, 96, 97 and 144.)

[115] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL `https://doi.org/10.1145/2025113.2025179`. (Cited on page 212.)

[116] Jie Fu and Ufuk Topcu. Probably approximately correct MDP learning and control with temporal logic constraints. In Dieter Fox, Lydia E. Kavraki, and Hanna Kurniawati, editors, *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014. ISBN 978-0-9923747-0-9. doi: 10.15607/RSS.2014.X.039. URL `http://www.roboticsproceedings.org/rss10/p39.html`. (Cited on pages 109, 115 and 211.)

[117] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. doi: 10.1109/32.87284. URL `https://doi.org/10.1109/32.87284`. (Cited on pages 27, 43, 56, 58, 68, 70, 77, 79, 87, 89, 208 and 216.)

[118] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining behavior models from user-intensive web applications. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 277–287. ACM, 2014. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568234. URL `https://doi.org/10.1145/2568225.2568234`. (Cited on page 209.)

[119] Georgios Giantamidis and Stavros Tripakis. Learning Moore machines from input-output traces. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 291–309, 2016. ISBN 978-3-319-48988-9. doi: 10.1007/978-3-319-48989-6_18. URL `https://doi.org/10.1007/978-3-319-48989-6_18`. (Cited on page 4.)

[120] Arthur Gill. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, 1962. (Cited on page 26.)

[121] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 213–224. ACM, 2016. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931061. URL `https://doi.org/10.1145/2931037.2931061`. (Cited on page 205.)

[122] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447 – 474, 1967. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(67)91165-5. URL `http://www.sciencedirect.com/science/article/pii/S0019995867911655`. (Cited on pages 162 and 220.)

[123] Rodolfo Gómez. A compositional translation of timed automata with deadlines to Uppaal timed automata. In Joël Ouaknine and Frits W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, volume 5813 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2009. ISBN 978-3-642-04367-3. doi: 10.1007/978-3-642-04368-0_15. URL `https://doi.org/10.1007/978-3-642-04368-0_15`. (Cited on page 165.)

[124] Wolfgang Grieskamp and Nicolas Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In Jon Whittle, Leif Geiger, and Michael Meisinger, editors, *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, Shanghai, China, May 27, 2006*, pages 59–66. ACM, 2006. ISBN 1-59593-394-8. doi: 10.1145/1138953.1138966. URL `https://doi.org/10.1145/1138953.1138966`. (Cited on pages 41 and 49.)

[125] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2006. ISBN 3-540-37376-4. doi: 10.1007/11817949_29. URL `https://doi.org/10.1007/11817949_29`. (Cited on pages 53, 161, 162, 210 and 211.)

[126] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010. doi: 10.1016/j.tcs.2010.07.008. URL `https://doi.org/10.1016/j.tcs.2010.07.008`. (Cited on pages 4, 8, 53, 161, 162, 210 and 211.)

[127] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. AMC: An adaptive model checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 2002. ISBN 3-540-43997-8. doi: 10.1007/3-540-45657-0_44. URL `https://doi.org/10.1007/3-540-45657-0_44`. (Cited on page 204.)

[128] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. Adaptive model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2002. ISBN 3-540-43419-4. doi: 10.1007/3-540-46002-0_25. URL `https://doi.org/10.1007/3-540-46002-0_25`. (Cited on page 204.)

[129] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006. doi: 10.1093/jigpal/jzl007. URL `https://doi.org/10.1093/jigpal/jzl007`. (Cited on pages 181 and 204.)

[130] Bernhard Großwindhager, Astrid Rupp, Martin Tappler, Markus Tranninger, Samuel Weiser, Bernhard Aichernig, Carlo Alberto Boano, Martin Horn, Gernot Kubin, Stefan Mangard, Martin Steinberger, and Kay Römer. Dependable internet of things for networked cars. *International Journal of Computing*, 16(4):226–237, 2017. ISSN 2312-5381. URL `http://computingonline.net/computing/article/view/911`. (Cited on page 12.)

[131] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. Modular system verification by inference, testing and reachability analysis. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 2008. ISBN 978-3-540-68514-2. doi: 10.1007/978-3-540-68524-1_16. URL `https://doi.org/10.1007/978-3-540-68524-1_16`. (Cited on page 206.)

[132] Roland Groz, Nicolas Brémond, and Adenilso Simão. Using adaptive sequences for learning non-resettable FSMs. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 30–43. PMLR, 2018. URL `http://proceedings.mlr.press/v93/groz19a.html`. (Cited on page 208.)

[133] Tobias R. Gundersen, Florian Lorber, Ulrik Nyman, and Christian Ovesen. Effortless fault localisation: Conformance testing of real-time systems in Ecdar. In Andrea Orlandini and Martin Zimmermann, editors, *Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018.*, volume 277 of *EPTCS*, pages 147–160, 2018. doi: 10.4204/EPTCS.277.11. URL `https://doi.org/10.4204/EPTCS.277.11`. (Cited on page 214.)

[134] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2002. ISBN 3-540-43353-8. doi: 10.1007/3-540-45923-5_6. URL `https://doi.org/10.1007/3-540-45923-5_6`. (Cited on pages 4 and 206.)

[135] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996. ISBN 0-8186-7463-6. doi: 10.1109/LICS.1996.561342. URL `https://doi.org/10.1109/LICS.1996.561342`. (Cited on page 189.)

[136] Anders Hessel and Paul Pettersson. CoVer – a test case generation tool for real-time systems. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems: Work-in-Progress and Position Papers, Tool Demonstrations, and Tutorial Abstracts of TestCom/FATES 2007*, pages 31–34, 2007. (Cited on page 214.)

[137] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2003. ISBN 3-540-20894-1. doi: 10.1007/978-3-540-24617-6_9. URL `https://doi.org/10.1007/978-3-540-24617-6_9`. (Cited on pages 162, 163, 164, 165 and 176.)

[138] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008. ISBN 978-3-540-78916-1. doi: 10.1007/978-3-540-78917-8_3. URL `https://doi.org/10.1007/978-3-540-78917-8_3`. (Cited on pages 174 and 214.)

[139] Kassel Hingee and Marcus Hutter. Equivalence of probabilistic tournament and polynomial ranking selection. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China*, pages 564–571. IEEE, 2008. doi: 10.1109/CEC.2008.4630852. URL `https://doi.org/10.1109/CEC.2008.4630852`. (Cited on page 170.)

[140] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL `http://doi.acm.org/10.1145/363235.363259`. (Cited on pages 1 and 2.)

[141] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`. (Cited on page 197.)

[142] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. ISSN 01621459. doi: 10.2307/2282952. URL `http://www.jstor.org/stable/2282952`. (Cited on pages 138, 146 and 209.)

[143] Karim Hossen, Roland Groz, Catherine Oriat, and Jean-Luc Richier. Automatic model inference of web applications for security testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*, pages 22–23. IEEE Computer Society, 2014. ISBN 978-0-7695-5194-4. doi: 10.1109/ICSTW.2014.47. URL `https://doi.org/10.1109/ICSTW.2014.47`. (Cited on page 205.)

[144] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS – lessons learned in the ZULU challenge. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010. ISBN 978-3-642-16557-3. doi: 10.1007/978-3-642-16558-0_55. URL `https://doi.org/10.1007/978-3-642-16558-0_55`. (Cited on pages 6, 29, 30, 57, 58, 68 and 208.)

[145] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011. ISBN 978-3-642-18274-7. doi: 10.1007/978-3-642-18275-4_19. URL `https://doi.org/10.1007/978-3-642-18275-4_19`. (Cited on pages 31 and 37.)

[146] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proceedings of the 16th annual IEEE International Conference on Network Protocols, 2008. ICNP 2008, Orlando, Florida, USA, 19-22 October 2008*, pages 114–123. IEEE Computer Society, 2008. ISBN 978-1-4244-2506-8. doi: 10.1109/ICNP.2008.4697030. URL `https://doi.org/10.1109/ICNP.2008.4697030`. (Cited on pages 205 and 207.)

[147] John Hughes, Ulf Norell, Nicholas Smallbone, and Thomas Arts. Find more bugs with QuickCheck! In Christof J. Budnik, Gordon Fraser, and Francesca Lonetti, editors, *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, pages 71–77. ACM, 2016. ISBN 978-1-4503-4151-6. doi: 10.1145/2896921.2896928. URL `http://doi.acm.org/10.1145/2896921.2896928`. (Cited on page 50.)

[148] Pham Ngoc Hung and Takuya Katayama. Modular conformance testing and assume-guarantee verification for evolving component-based software. In *15th Asia-Pacific Software Engineering*

Conference (APSEC 2008), 3-5 December 2008, Beijing, China, pages 479–486. IEEE Computer Society, 2008. ISBN 978-0-7695-3446-6. doi: 10.1109/APSEC.2008.51. URL `https://doi.org/10.1109/APSEC.2008.51`. (Cited on page 205.)

[149] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003. ISBN 3-540-40524-0. doi: 10.1007/978-3-540-45069-6_31. URL `https://doi.org/10.1007/978-3-540-45069-6_31`. (Cited on pages 51 and 206.)

[150] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064. URL `https://dx.doi.org/10.1109/IEEESTD.1990.101064`. (Cited on pages 2 and 43.)

[151] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014. ISBN 978-3-319-11163-6. doi: 10.1007/978-3-319-11164-3_26. URL `https://doi.org/10.1007/978-3-319-11164-3_26`. (Cited on pages 17, 20, 24, 30, 43, 51, 62, 79, 83, 140, 171, 208 and 216.)

[152] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib – a framework for active automata learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_32. URL `https://doi.org/10.1007/978-3-319-21690-4_32`. (Cited on pages 15, 18, 26, 27, 28, 37, 42, 50, 52, 58, 70, 79, 195, 206, 207 and 208.)

[153] ISO/IEC 20922:2016. Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1, ISO/IEC 20922:2016. Standard, International Organization for Standardization, Geneva, CH, June 2016. (Cited on pages 6 and 34.)

[154] Claude Jard and Thierry Jéron. TGV: Theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005. doi: 10.1007/s10009-004-0153-x. URL `https://doi.org/10.1007/s10009-004-0153-x`. (Cited on pages 162 and 213.)

[155] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62. URL `https://doi.org/10.1109/TSE.2010.62`. (Cited on pages 57, 61, 63, 65, 66 and 78.)

[156] Colin G. Johnson. Genetic programming with fitness based on model checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Esparcia-Alcázar, editors, *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124. Springer, 2007. ISBN 978-3-540-71602-0. doi: 10.1007/978-3-540-71605-1_11. URL `https://doi.org/10.1007/978-3-540-71605-1_11`. (Cited on page 212.)

[157] Bengt Jonsson and Frits W. Vaandrager. Learning Mealy machines with timers. Online preprint, 2018. Available via `http://www.sws.cs.ru.nl/publications/papers/fvaan/MMT/`, accessed on November 4, 2019. (Cited on pages 161 and 211.)

[158] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. ISBN 0-306-30707-3. doi: 10.1007/978-1-4684-2001-2_9. URL `https://doi.org/10.1007/978-1-4684-2001-2_9`. (Cited on page 60.)

[159] Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer*, 19(4):449–464, 2017. doi: 10.1007/s10009-016-0418-1. URL `https://doi.org/10.1007/s10009-016-0418-1`. (Cited on pages 8, 162 and 212.)

[160] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994. ISBN 978-0-262-11193-5. URL `https://mitpress.mit.edu/books/introduction-computational-learning-theory`. (Cited on pages 23, 79, 195 and 204.)

[161] Ali Khalili and Armando Tacchella. Learning nondeterministic Mealy machines. In Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka, editors, *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014.*, volume 34 of *JMLR Workshop and Conference Proceedings*, pages 109–123. JMLR.org, 2014. URL `http://proceedings.mlr.press/v34/khalili14a.html`. (Cited on pages 4, 7, 52, 91, 92, 98, 209 and 219.)

[162] Jin Hyun Kim, Kim Guldstrand Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. Formal analysis and testing of real-time automotive systems using UPPAAL tools. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2015. ISBN 978-3-319-19457-8. doi: 10.1007/978-3-319-19458-5_4. URL `https://doi.org/10.1007/978-3-319-19458-5_4`. (Cited on page 214.)

[163] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`. (Cited on page 197.)

[164] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey M. Voas. DDoS in the IoT: Mirai and other botnets. *IEEE Computer*, 50(7):80–84, 2017. doi: 10.1109/MC.2017.201. URL `https://doi.org/10.1109/MC.2017.201`. (Cited on page 1.)

[165] Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. Learning probabilistic systems from tree samples. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 441–450. IEEE Computer Society, 2012. ISBN 978-1-4673-2263-8. doi: 10.1109/LICS.2012.54. URL `https://doi.org/10.1109/LICS.2012.54`. (Cited on page 210.)

[166] I. Koufareva, Alexandre Petrenko, and Nina Yevtushenko. Test generation driven by user-defined fault models. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Methods and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems, September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 215–236. Kluwer, 1999. ISBN 0-7923-8581-0. doi: 10.1007/978-0-387-35567-2_14. URL `https://doi.org/10.1007/978-0-387-35567-2_14`. (Cited on page 66.)

[167] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6. (Cited on pages 8, 10, 163 and 165.)

[168] Willibald Krenn, Dejan Nickovic, and Loredana Tec. Incremental language inclusion checking for networks of timed automata. In Víctor A. Braberman and Laurent Fribourg, editors, *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, volume 8053 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013. ISBN 978-3-642-40228-9. doi: 10.1007/978-3-642-40229-6_11. URL `https://doi.org/10.1007/978-3-642-40229-6_11`. (Cited on page 214.)

[169] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. Momut::UML model-based mutation testing for UML. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–8. IEEE Computer Society, 2015. ISBN 978-1-4799-7125-1. doi: 10.1109/ICST.2015.7102627. URL `https://doi.org/10.1109/ICST.2015.7102627`. (Cited on pages 7, 213 and 220.)

[170] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009. doi: 10.1007/s10703-009-0065-1. URL `https://doi.org/10.1007/s10703-009-0065-1`. (Cited on pages 162 and 214.)

[171] Sebastian Kunze, Wojciech Mostowski, Mohammad Reza Mousavi, and Mahsa Varshosaz. Generation of failure models through automata learning. In *2016 Workshop on Automotive Systems/-Software Architectures (WASA'16)*, pages 22–25. IEEE, April 2016. doi: 10.1109/WASA.2016.7. URL `https://doi.org/10.1109/WASA.2016.7`. (Cited on page 206.)

[172] Marta Z. Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 5–22. Springer, 2013. ISBN 978-3-319-02443-1. doi: 10.1007/978-3-319-02444-8_2. URL `http://dx.doi.org/10.1007/978-3-319-02444-8_2`. (Cited on page 95.)

[173] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Analysis of a gossip protocol in PRISM. *SIGMETRICS Performance Evaluation Review*, 36(3):17–22, 2008. doi: 10.1145/1481506.1481511. URL `https://doi.org/10.1145/1481506.1481511`. (Cited on pages 92 and 98.)

[174] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. ISBN 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_47. URL `http://dx.doi.org/10.1007/978-3-642-22110-1_47`. (Cited on pages 96, 100, 101, 102, 103, 110, 116, 152, 153, 155 and 209.)

[175] Zhifeng Lai, S. C. Cheung, and Yunfei Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. In *Sixth International Conference on Quality Software (QSIC 2006), 26-28 October 2006, Beijing, China*, pages 410–417. IEEE Computer Society, 2006. ISBN 0-7695-2718-3. doi: 10.1109/QSIC.2006.25. URL `https://doi.org/10.1109/QSIC.2006.25`. (Cited on pages 205 and 211.)

[176] Kim Guldstrand Larsen. Verification and performance analysis of embedded and cyber-physical systems using UPPAAL. In César Benavente-Peces, Andreas Ahrens, and Joaquim Filipe, editors, *PECCS 2014 - Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems, Lisbon, Portugal, 7-9 January, 2014*. SciTePress, 2014. ISBN 978-989-758-000-0. URL `http://www.peccs.org/KeynoteSpeakers.aspx?y=2014`. Keynote Lecture. (Cited on page 189.)

[177] Kim Guldstrand Larsen and Axel Legay. Statistical model checking: Past, present, and future. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 3–15, 2016. ISBN 978-3-319-47165-5. doi: 10.1007/978-3-319-47166-2_1. URL `http://dx.doi.org/10.1007/978-3-319-47166-2_1`. (Cited on pages 97, 100, 101, 102 and 103.)

[178] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. doi: 10.1007/s100090050010. URL `https://doi.org/10.1007/s100090050010`. (Cited on pages 162, 164, 174 and 214.)

[179] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2004. ISBN 3-540-25109-X. doi: 10.1007/978-3-540-31848-4_6. URL `https://doi.org/10.1007/978-3-540-31848-4_6`. (Cited on page 122.)

[180] Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. Safe and optimal adaptive cruise control. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, volume 9360 of *Lecture Notes in Computer Science*, pages 260–277. Springer, 2015. ISBN 978-3-319-23505-9. doi: 10.1007/978-3-319-23506-6_17. URL `https://doi.org/10.1007/978-3-319-23506-6_17`. (Cited on page 212.)

[181] Kim Guldstrand Larsen, Florian Lorber, Brian Nielsen, and Ulrik Nyman. Mutation-based test-case generation with Ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 319–328. IEEE Computer Society, 2017. ISBN 978-1-5090-6676-6. doi: 10.1109/ICSTW.2017.60. URL `https://doi.org/10.1109/ICSTW.2017.60`. (Cited on page 214.)

[182] Kim Guldstrand Larsen, Adrien Le Coënt, Marius Mikucionis, and Jakob Haahr Taankvist. Guaranteed control synthesis for continuous systems in Uppaal Tiga. In Roger D. Chamberlain, Walid Taha, and Martin Törngren, editors, *Cyber Physical Systems. Model-Based Design - 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Turin, Italy, October 4-5, 2018, Revised Selected Papers*, volume 11615 of *Lecture Notes in Computer Science*, pages 113–133. Springer, 2018. ISBN 978-3-030-23702-8. doi: 10.1007/978-3-030-23703-5_6. URL `https://doi.org/10.1007/978-3-030-23703-5_6`. (Cited on page 212.)

[183] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994. doi: 10.1109/12.272431. URL `https://doi.org/10.1109/12.272431`. (Cited on pages 30, 57, 68 and 208.)

[184] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996. ISSN 0018-9219. doi: 10.1109/5.533956. URL `https://doi.org/10.1109/5.533956`. (Cited on pages 25, 28, 56 and 213.)

[185] Edward A. Lee. Cyber physical systems: Design challenges. In *ISORC 2018, 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 363–369. IEEE, 2008. doi: 10.1109/ISORC.2008.25. URL `http://doi.org/10.1109/ISORC.2008.25`. (Cited on page 189.)

[186] Raluca Lefticaru, Florentin Ipate, and Cristina Tudose. Automated model design using genetic algorithms and model checking. In Petros Kefalas, Demosthenes Stamatis, and Christos Douligeris, editors, *2009 Fourth Balkan Conference in Informatics, BCI 2009, Thessaloniki, Greece, 17-19 September 2009*, pages 79–84. IEEE Computer Society, 2009. ISBN 978-0-7695-3783-2. doi: 10.1109/BCI.2009.15. URL `https://doi.org/10.1109/BCI.2009.15`. (Cited on page 212.)

[187] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010. ISBN 978-3-642-16611-2. doi: 10.1007/978-3-642-16612-9_11. URL `https://doi.org/10.1007/978-3-642-16612-9_11`. (Cited on page 100.)

[188] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Scalable verification of Markov decision processes. In Carlos Canal and Akram Idani, editors, *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, volume 8938 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2014. ISBN 978-3-319-15200-4. doi: 10.1007/978-3-319-15201-1_23. URL `http://dx.doi.org/10.1007/978-3-319-15201-1_23`. (Cited on page 211.)

[189] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940. URL `https://doi.org/10.1109/MC.1993.274940`. (Cited on page 1.)

[190] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In Phil McMinn, editor, *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom*, pages 59–70. IEEE Computer Society, 2006. ISBN 0-7695-2672-1. doi: 10.1109/TAIC-PART.2006.15. URL `https://doi.org/10.1109/TAIC-PART.2006.15`. (Cited on page 206.)

[191] Shang-Wei Lin, Étienne André, Jin Song Dong, Jun Sun, and Yang Liu. An efficient algorithm for learning event-recording automata. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 463–472. Springer, 2011. ISBN 978-3-642-24371-4. doi: 10.1007/978-3-642-24372-1_35. URL `https://doi.org/10.1007/978-3-642-24372-1_35`. (Cited on page 211.)

[192] Shang-Wei Lin, Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Learning assumptions for compositional verification of timed systems. *IEEE Transactions on Software Engineering*, 40(2): 137–153, 2014. doi: 10.1109/TSE.2013.57. URL `https://doi.org/10.1109/TSE.2013.57`. (Cited on page 211.)

[193] Florian Lorber, Kim Guldstrand Larsen, and Brian Nielsen. Model-based mutation testing of real-time systems via model checking. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 59–68. IEEE Computer Society, 2018. ISBN 978-1-5386-6352-3. doi: 10.1109/ICSTW.2018.00029. URL `https://doi.org/10.1109/ICSTW.2018.00029`. (Cited on page 214.)

[194] Simon M. Lucas and T. Jeff Reynolds. Learning DFA: Evolution versus evidence driven state merging. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, 8 - 12 December 2003, Canberra, Australia*, pages 351–358. IEEE, 2003. doi: 10.1109/CEC.2003.1299597. URL `https://doi.org/10.1109/CEC.2003.1299597`. (Cited on page 211.)

[195] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995. doi: 10.1006/inco.1995.1070. URL `https://doi.org/10.1006/inco.1995.1070`. (Cited on page 22.)

[196] Zohar Manna and Amir Pnueli. Verifying hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems. HS 1992, HS 1991*, volume 736 of *Lecture Notes in Computer Science*, pages 4–35. Springer, 1992. ISBN 3-540-57318-6. doi: 10.1007/3-540-57318-6_22. URL `https://doi.org/10.1007/3-540-57318-6_22`. (Cited on pages 5 and 189.)

[197] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning probabilistic automata for model checking. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 111–120. IEEE Computer Society, 2011. ISBN 978-1-4577-0973-9. doi: 10.1109/QEST.2011.21. URL `http://dx.doi.org/10.1109/QEST.2011.21`. (Cited on pages 94 and 209.)

[198] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning Markov decision processes for model checking. In Uli Fahrenberg, Axel Legay, and Claus R. Thrane, editors, *Proceedings Quantities in Formal Methods, QFM 2012, Paris, France, 28 August 2012.*, volume 103 of *EPTCS*, pages 49–63, 2012. doi: 10.4204/EPTCS.103.6. URL `http://dx.doi.org/10.4204/EPTCS.103.6`. (Cited on pages 4, 92, 95, 100, 101, 103, 107, 115, 118, 121, 125, 126, 156 and 209.)

[199] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016. doi: 10.1007/s10994-016-5565-9. URL `https://doi.org/10.1007/s10994-016-5565-9`. (Cited on pages 4, 7, 52, 92, 93, 94, 95, 96, 100, 101, 103, 107, 109, 110, 111, 112, 121, 125, 126, 138, 146, 152, 153, 156, 158, 161, 171, 209, 210, 218 and 221.)

[200] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *Ninth IEEE International High-Level Design Validation and Test Workshop 2004, Sonoma Valley, CA, USA, November 10-12, 2004*, pages 95–100. IEEE Computer Society, 2004. ISBN 0-7803-8714-7. doi: 10.1109/HLDVT.2004.1431246. URL `https://doi.org/10.1109/HLDVT.2004.1431246`. (Cited on pages 4, 15, 18 and 209.)

[201] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: A tool for automatic black-box testing. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1013–1015. ACM, 2011. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985979. URL `https://doi.org/10.1145/1985793.1985979`. (Cited on page 207.)

[202] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: Automatic black-box testing of interactive applications. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 81–90. IEEE Computer Society, 2012. ISBN 978-1-4577-1906-6. doi: 10.1109/ICST.2012.88. URL `https://doi.org/10.1109/ICST.2012.88`. (Cited on page 207.)

[203] Ramy Medhat, S. Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. A framework for mining hybrid automata from input/output traces. In Alain Girault and Nan Guan, editors, *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 177–186. IEEE, 2015. ISBN 978-1-4673-8079-9. doi: 10.1109/EMSOFT.2015.7318273. URL `https://doi.org/10.1109/EMSOFT.2015.7318273`. (Cited on page 190.)

[204] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. Improved learning for stochastic timed models by state-merging algorithms. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 178–193, 2017. ISBN 978-3-319-57287-1. doi: 10.1007/978-3-319-57288-8_13. URL `https://doi.org/10.1007/978-3-319-57288-8_13`. (Cited on page 210.)

[205] Karl Meinke. Learning-based testing of cyber-physical systems-of-systems: A platooning study. In Philipp Reinecke and Antinisca Di Marco, editors, *Computer Performance Engineering - 14th European Workshop, EPEW 2017, Berlin, Germany, September 7-8, 2017, Proceedings*, volume 10497 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2017. ISBN 978-3-319-66582-5. doi: 10.1007/978-3-319-66583-2_9. URL `https://doi.org/10.1007/978-3-319-66583-2_9`. (Cited on pages 201 and 212.)

[206] Karl Meinke and Muddassar A. Sindhu. Incremental learning-based testing for reactive systems. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2011. ISBN 978-3-642-21767-8. doi: 10.1007/978-3-642-21768-5_11. URL `https://doi.org/10.1007/978-3-642-21768-5_11`. (Cited on pages 41 and 205.)

[207] Karl Meinke and Muddassar A. Sindhu. LBTest: A learning-based testing tool for reactive systems. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 447–454. IEEE Computer Society, 2013. ISBN 978-1-4673-5961-0. doi: 10.1109/ICST.2013.62. URL `https://doi.org/10.1109/ICST.2013.62`. (Cited on page 205.)

[208] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, volume 336 of *Communications in Computer and Information Science*, pages 248–260. Springer, 2011. ISBN 978-3-642-34780-1. doi: 10.1007/978-3-642-34781-8_19. URL `https://doi.org/10.1007/978-3-642-34781-8_19`. (Cited on page 50.)

[209] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998. ISBN 978-0-262-63185-3. (Cited on pages 165, 166 and 168.)

[210] Joshua Moerman. Yannakakis – test-case generator, 2015. `https://gitlab.science.ru.nl/moerman/Yannakakis`, accessed on November 4, 2019. (Cited on page 68.)

[211] Morteza Mohaqeqi, Mohammad Reza Mousavi, and Walid Taha. Conformance testing of cyber-physical systems: A comparative study. *Electronic Communications of the EASST*, 70, 2014. doi: 10.14279/tuj.eceasst.70.982. URL `https://doi.org/10.14279/tuj.eceasst.70.982`. (Cited on page 189.)

[212] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN 026201825X, 9780262018258. (Cited on page 29.)

[213] MQTT. MQTT website. `http://mqtt.org/`, 2019. accessed on November 4, 2019. (Cited on page 35.)

[214] Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In Tiziana Margaria, Susanne Graf, and Kim Guldstrand Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 390–416. Springer, 2018. ISBN 978-3-030-22347-2. doi: 10.1007/978-3-030-22348-9_23. URL `https://doi.org/10.1007/978-3-030-22348-9_23`. (Cited on pages 9, 58, 69 and 79.)

[215] Laura Nenzi, Simone Silvetti, Ezio Bartocci, and Luca Bortolussi. A robust genetic algorithm for learning temporal specifications from data. In Annabelle McIver and Andras Horvath, editors, *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*, volume 11024 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2018. ISBN 978-3-319-99153-5. doi: 10.1007/978-3-319-99154-2_20. URL `https://doi.org/10.1007/978-3-319-99154-2_20`. (Cited on pages 211 and 212.)

[216] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 0002-9939, 1088-6826/e. doi: 10.2307/2033204. URL `https://www.jstor.org/stable/2033204`. (Cited on pages 21, 127 and 128.)

[217] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59–77, 2003. doi: 10.1007/s10009-002-0094-1. URL `https://doi.org/10.1007/s10009-002-0094-1`. (Cited on page 214.)

[218] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, Dortmund University of Technology, 2003. URL `https://d-nb.info/969717474/34`. (Cited on page 18.)

[219] Oliver Niggemann, Benno Stein, Asmir Vodencarevic, Alexander Maier, and Hans Kleine Büning. Learning behavior models for hybrid timed systems. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4993`. (Cited on page 190.)

[220] Gethin Norman and Vitaly Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006. doi: 10.3233/JCS-2006-14604. URL `http://content.iospress.com/articles/journal-of-computer-security/jcs268`. (Cited on pages 92 and 98.)

[221] Ayoub Nouri, Balaji Raman, Marius Bozga, Axel Legay, and Saddek Bensalem. Faster statistical model checking by means of abstraction and learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014,*

Toronto, ON, Canada, September 22-25, 2014. Proceedings, volume 8734 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2014. ISBN 978-3-319-11163-6. doi: 10.1007/978-3-319-11164-3_28. URL `https://doi.org/10.1007/978-3-319-11164-3_28`. (Cited on page 209.)

[222] Mariusz Nowostawski and Riccardo Poli. Parallel genetic algorithm taxonomy. In Lakhmi C. Jain, editor, *Third International Conference on Knowledge-Based Intelligent Information Engineering Systems, KES 1999, Adelaide, South Australia, 31 August - 1 September 1999, Proceedings*, pages 88–92. IEEE, 1999. ISBN 0-7803-5578-4. doi: 10.1109/KES.1999.820127. URL `https://doi.org/10.1109/KES.1999.820127`. (Cited on page 166.)

[223] Masashi Okamoto. Some inequalities relating to the partial sum of binomial probabilities. *Annals of the Institute of Statistical Mathematics*, 10(1):29–35, 1959. ISSN 1572-9052. doi: 10.1007/BF02883985. URL `http://dx.doi.org/10.1007/BF02883985`. (Cited on pages 97, 103, 106, 110, 118 and 123.)

[224] Sean O'Kane. The Verge: Tesla owner discovers problem with 'dog mode' air conditioning feature. Available via `https://www.theverge.com/2019/8/1/20750085/tesla-dog-mode-flaw-elon-musk-software-update`, news article, 2019. accessed on November 4, 2019. (Cited on page 2.)

[225] José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992. doi: 10.1142/9789812797919_0007. URL `https://doi.org/10.1142/9789812797919_0007`. (Cited on page 4.)

[226] Martijn Oostdijk, Vlad Rusu, Jan Tretmans, René G. de Vries, and Tim A. C. Willemse. Integrating verification, testing, and learning for cryptographic protocols. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, pages 538–557. Springer, 2007. ISBN 978-3-540-73209-9. doi: 10.1007/978-3-540-73210-5_28. URL `https://doi.org/10.1007/978-3-540-73210-5_28`. (Cited on page 205.)

[227] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 401–411. IEEE Computer Society, 2017. ISBN 978-1-5090-6031-3. doi: 10.1109/ICST.2017.43. URL `https://doi.org/10.1109/ICST.2017.43`. (Cited on page 210.)

[228] Ron Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005. ISBN 0672327988. (Cited on page 2.)

[229] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999. ISBN 0-7923-8646-9. doi: 10.1007/978-0-387-35578-8_13. URL `https://doi.org/10.1007/978-0-387-35578-8_13`. (Cited on page 204.)

[230] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002. doi: 10.25596/jalc-2002-225. URL `https://doi.org/10.25596/jalc-2002-225`. (Cited on pages 56, 99, 100, 204, 205, 207 and 208.)

[231] Alexandre Petrenko and Florent Avellaneda. Learning communicating state machines. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs - 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, volume 11823 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2019. ISBN 978-3-030-31156-8. doi: 10.1007/978-3-030-31157-5_8. URL `https://doi.org/10.1007/978-3-030-31157-5_8`. (Cited on page 121.)

[232] Alexandre Petrenko, Keqin Li, Roland Groz, Karim Hossen, and Catherine Oriat. Inferring approximated models for systems engineering. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 249–253. IEEE Computer Society, 2014. ISBN 978-1-4799-3465-2. doi: 10.1109/HASE.2014.46. URL `https://doi.org/10.1109/HASE.2014.46`. (Cited on page 205.)

[233] Alexandre Petrenko, Omer Nguena-Timo, and S. Ramesh. Multiple mutation testing from FSM. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2016. ISBN 978-3-319-39569-2. doi: 10.1007/978-3-319-39570-8_15. URL `https://doi.org/10.1007/978-3-319-39570-8_15`. (Cited on page 66.)

[234] Andrea Pferscher. Active model learning of timed automata via genetic programming, 2019. Master's thesis, Graz University of Technology. (Cited on pages 10, 12, 162, 179, 180, 181, 183, 184, 187, 217 and 219.)

[235] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL `http://www.rfc-editor.org/rfc/rfc793.txt`. Internet Requests for Comments. (Cited on pages 8 and 30.)

[236] Alexander Pretschner. Defect-based testing. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 224–245. IOS Press, 2015. ISBN 978-1-61499-494-7. doi: 10.3233/978-1-61499-495-4-224. URL `http://dx.doi.org/10.3233/978-1-61499-495-4-224`. (Cited on page 61.)

[237] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with Webtest. In Tevfik Bultan and Tao Xie, editors, *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), TAV-WEB 2008, Seattle, Washington, USA, July 21, 2008*, pages 1–7. ACM, 2008. ISBN 978-1-60558-053-1. doi: 10.1145/1390832.1390833. URL `https://doi.org/10.1145/1390832.1390833`. (Cited on page 208.)

[238] Brian Randell. Fifty years of software engineering - or - the view from Garmisch. *CoRR*, abs/1805.02742, 2018. URL `http://arxiv.org/abs/1805.02742`. (Cited on pages 1 and 215.)

[239] Adnan Rashid, Umair Siddique, and Osman Hasan. Formal verification of platoon control strategies. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, volume 10886 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2018. ISBN 978-3-319-92969-9. doi: 10.1007/978-3-319-92970-5_14. URL `https://doi.org/10.1007/978-3-319-92970-5_14`. (Cited on page 212.)

[240] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993. doi: 10.1006/inco.1993.1021. URL `https://doi.org/10.1006/inco.1993.1021`. (Cited on pages 22, 23, 24, 29, 30, 57, 58, 62, 68, 70, 79, 83, 131, 140, 208 and 216.)

[241] Krishan K. Sabnani and Anton T. Dahbura. A protocol test generation procedure. *Computer Networks*, 15(4):285–297, 1988. doi: 10.1016/0169-7552(88)90064-5. URL `https://doi.org/10.1016/0169-7552(88)90064-5`. (Cited on page 28.)

[242] Indranil Saha and Debapriyay Mukhopadhyay. Quantitative analysis of a probabilistic non-repudiation protocol through model checking. In Atul Prakash and Indranil Gupta, editors, *Information Systems Security, 5th International Conference, ICISS 2009, Kolkata, India, December 14-18, 2009, Proceedings*, volume 5905 of *Lecture Notes in Computer Science*, pages 292–300. Springer, 2009. ISBN 978-3-642-10771-9. doi: 10.1007/978-3-642-10772-6_22. URL `https://doi.org/10.1007/978-3-642-10772-6_22`. (Cited on page 98.)

[243] Richard Schumi, Priska Lang, Bernhard K. Aichernig, Willibald Krenn, and Rupert Schlick. Checking response-time properties of web-service applications under stochastic user profiles. In Nina Yevtushenko, Ana Rosa Cavalli, and Hüsnü Yenigün, editors, *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*, volume 10533 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2017. ISBN 978-3-319-67548-0. doi: 10.1007/978-3-319-67549-7_18. URL `https://doi.org/10.1007/978-3-319-67549-7_18`. (Cited on page 211.)

[244] Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2016. ISBN 978-3-319-33692-3. doi: 10.1007/978-3-319-33693-0_20. URL `https://doi.org/10.1007/978-3-319-33693-0_20`. (Cited on page 206.)

[245] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2004. ISBN 3-540-22342-8. doi: 10.1007/978-3-540-27813-9_16. URL `https://doi.org/10.1007/978-3-540-27813-9_16`. (Cited on page 209.)

[246] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_14. URL `https://doi.org/10.1007/978-3-642-05089-3_14`. (Cited on pages 4, 18, 20, 21, 56, 79 and 209.)

[247] Muzammil Shahbaz and Roland Groz. Analysis and testing of black-box component-based systems by inferring partial models. *Software Testing, Verification & Reliability*, 24(4):253–288, 2014. doi: 10.1002/stvr.1491. URL `https://doi.org/10.1002/stvr.1491`. (Cited on page 206.)

[248] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning parameterized state machine model for integration testing. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 2*, pages 755–760. IEEE

Computer Society, 2007. doi: 10.1109/COMPSAC.2007.134. URL https://doi.org/10.1109/COMPSAC.2007.134. (Cited on page 206.)

[249] Muzammil Shahbaz, Benoît Parreaux, and Francis Klay. Model inference approach for detecting feature interactions in integrated systems. In Lydie du Bousquet and Jean-Luc Richier, editors, *Ninth International Conference of Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France*, pages 161–171. IOS Press, 2007. ISBN 978-1-58603-845-8. URL http://ebooks.iospress.nl/publication/29339. (Cited on page 206.)

[250] Guoqiang Shu and David Lee. Testing security properties of protocol implementations – a machine learning based approach. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 25. IEEE Computer Society, 2007. ISBN 0-7695-2837-6. doi: 10.1109/ICDCS.2007.147. URL https://doi.org/10.1109/ICDCS.2007.147. (Cited on pages 205 and 207.)

[251] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5048 of *Lecture Notes in Computer Science*, pages 299–304. Springer, 2008. ISBN 978-3-540-68854-9. doi: 10.1007/978-3-540-68855-6_19. URL https://doi.org/10.1007/978-3-540-68855-6_19. (Cited on pages 205 and 207.)

[252] T.W. Simpson, A.J. Booker, D. Ghosh, A.A. Giunta, P.N. Koch, and R.-J. Yang. Approximation methods in multidisciplinary analysis and optimization: A panel discussion. *Structural and Multidisciplinary Optimization*, 27(5):302–313, 2004. ISSN 1615-1488. doi: 10.1007/s00158-004-0389-9. URL https://doi.org/10.1007/s00158-004-0389-9. (Cited on page 190.)

[253] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 521–538. IEEE Computer Society, 2017. ISBN 978-1-5090-5533-3. doi: 10.1109/SP.2017.46. URL https://doi.org/10.1109/SP.2017.46. (Cited on pages 52 and 204.)

[254] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2015. ISBN 978-3-319-25422-7. doi: 10.1007/978-3-319-25423-4_5. URL https://doi.org/10.1007/978-3-319-25423-4_5. (Cited on pages 30, 40, 51, 57, 58, 68, 76, 78, 208 and 216.)

[255] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing model learning with mutation-based fuzzing. *CoRR*, abs/1611.02429, 2016. URL http://arxiv.org/abs/1611.02429. (Cited on page 208.)

[256] Jan Springintveld, Frits W. Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001. doi: 10.1016/S0304-3975(99)00134-6. URL https://doi.org/10.1016/S0304-3975(99)00134-6. (Cited on pages 164, 165, 168, 213 and 220.)

[257] Andy Stanford-Clark and Hong Linh Truong. MQTT For Sensor Networks (MQTT-SN) – Protocol Specification Version 1.2. Technical report, International Business Machines Corporation (IBM), November 2013. URL `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`. Available via `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`, accessed on November 4, 2019. (Cited on page 35.)

[258] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011. ISBN 978-3-642-21454-7. doi: 10.1007/978-3-642-21455-4_8. URL `https://doi.org/10.1007/978-3-642-21455-4_8`. (Cited on pages 20, 21, 23, 79, 85, 86, 127, 128, 130 and 209.)

[259] Martin Tappler. prob-black-reach – Java implementation of probabilistic black-box reachability checking [16, 18], 2017. `https://github.com/mtappler/prob-black-reach`, accessed on November 4, 2019. (Cited on pages 10 and 110.)

[260] Martin Tappler. mut-learn – randomised mutation-based equivalence testing for active automata learning [15, 17], 2017. `https://github.com/mtappler/mut-learn`, accessed on November 4, 2019. (Cited on pages 10 and 68.)

[261] Martin Tappler. Evaluation material for L^*-based learning of Markov decision processes [265, 266], 2019. Available via `https://doi.org/10.6084/m9.figshare.7960928.v1`, accessed on November 4, 2019. (Cited on pages 10, 125, 152 and 158.)

[262] Martin Tappler and Andrea Pferscher. Supplementary material for "Learning timed automata via genetic programming" [264] and "Time to learn – learning timed automata from tests" [267], 2019. Available via `https://figshare.com/articles/Supplementary_Material_for_Learning_Timed_Automata_via_Genetic_Programming_/5513575`, accessed on November 4, 2019. (Cited on pages 10, 163, 172 and 173.)

[263] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. Model-based testing IoT communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 276–287. IEEE Computer Society, 2017. ISBN 978-1-5090-6031-3. doi: 10.1109/ICST.2017.32. URL `https://doi.org/10.1109/ICST.2017.32`. (Cited on pages 6, 9, 10, 15, 33, 45, 46, 69, 73, 109, 113, 121, 154, 203 and 204.)

[264] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Learning timed automata via genetic programming. *CoRR*, abs/1808.07744, 2018. URL `http://arxiv.org/abs/1808.07744`. (Cited on pages 9, 11, 161, 203 and 252.)

[265] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim Guldstrand Larsen. L^*-based learning of Markov decision processes. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 651–669. Springer, 2019. ISBN 978-3-030-30941-1. doi: 10.1007/978-3-030-30942-8_38. URL `https://doi.org/10.1007/978-3-030-30942-8_38`. (Cited on pages 9, 12, 91, 92, 125, 203 and 252.)

[266] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim Guldstrand Larsen. L^*-based learning of Markov decision processes (extended version). *CoRR*, abs/1906.12239, 2019. URL `http://arxiv.org/abs/1906.12239`. (Cited on pages 9, 12, 91, 92, 125 and 252.)

[267] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn – learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 216–235. Springer, 2019. ISBN 978-3-030-29661-2. doi: 10.1007/978-3-030-29662-9_13. URL `https://doi.org/10.1007/978-3-030-29662-9_13`. (Cited on pages 9, 11, 161, 203 and 252.)

[268] Robert A. Thacker, Kevin R. Jones, Chris J. Myers, and Hao Zheng. Automatic abstraction for verification of cyber-physical systems. In Janos Sztipanovits and Raj Rajkumar, editors, *ACM/IEEE 1st International Conference on Cyber-Physical Systems, ICCPS '10, Stockholm, Sweden, April 12-15, 2010*, pages 12–21. ACM, 2010. ISBN 978-1-4503-0066-7. doi: 10.1145/1795194.1795197. URL `https://doi.org/10.1145/1795194.1795197`. (Cited on page 189.)

[269] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996. doi: 10.1016/S0169-7552(96)00017-7. URL `https://doi.org/10.1016/S0169-7552(96)00017-7`. (Cited on pages 24, 25, 98 and 165.)

[270] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. (Cited on pages 28, 38, 162, 209 and 214.)

[271] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. ISBN 978-3-540-78916-1. doi: 10.1007/978-3-540-78917-8_1. URL `https://doi.org/10.1007/978-3-540-78917-8_1`. (Cited on pages 3, 25, 26, 93 and 162.)

[272] Wen-Guey Tzeng. Learning probabilistic automata and Markov chains via queries. *Machine Learning*, 8:151–166, 1992. doi: 10.1007/BF00992862. URL `https://doi.org/10.1007/BF00992862`. (Cited on page 210.)

[273] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007. ISBN 978-0-12-372501-1. URL `http://www.elsevierdirect.com/product.jsp?isbn=9780123725011`. (Cited on page 2.)

[274] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability*, 22(5):297–312, 2012. doi: 10.1002/stvr.456. URL `https://doi.org/10.1002/stvr.456`. (Cited on pages 2, 39, 41 and 105.)

[275] Frits W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017. doi: 10.1145/2967606. URL `https://doi.org/10.1145/2967606`. (Cited on pages 18 and 19.)

[276] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984. doi: 10.1145/1968.1972. URL `https://doi.org/10.1145/1968.1972`. (Cited on pages 29, 126, 158 and 159.)

[277] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, Jul 1973. ISSN 1573-8337. doi: 10.1007/BF01068590. URL `https://doi.org/10.1007/BF01068590`. (Cited on pages 26, 29, 43, 53, 56, 57, 79, 99, 100, 204, 205, 208, 213 and 218.)

[278] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In Michel Wermelinger and Harald C. Gall, editors, *ESEC/FSE-13, Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, pages 273–282. ACM, 2005. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081751. URL `https://doi.org/10.1145/1081706.1081751`. (Cited on page 27.)

[279] Sicco Verwer, Mathijs De Weerdt, and Cees Witteveen. An algorithm for learning real-time automata. In Maarten van Someren, Sophia Katrenko, and Pieter Adriaans, editors, *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007*, pages 128–135, 2007. (Cited on pages 4, 8, 161, 162, 190 and 210.)

[280] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, volume 6339 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 2010. ISBN 978-3-642-15487-4. doi: 10.1007/978-3-642-15488-1_17. URL `http://dx.doi.org/10.1007/978-3-642-15488-1_17`. (Cited on pages 4, 161, 162, 210 and 221.)

[281] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. The efficiency of identifying timed automata and the power of clocks. *Information and Computation*, 209(3):606–625, 2011. doi: 10.1016/j.ic.2010.11.023. URL `https://doi.org/10.1016/j.ic.2010.11.023`. (Cited on page 53.)

[282] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, 86(3):295–333, 2012. doi: 10.1007/s10994-011-5265-4. URL `https://doi.org/10.1007/s10994-011-5265-4`. (Cited on pages 4 and 53.)

[283] Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. *Electronic Communications of the EASST*, 72, 2015. doi: 10.14279/tuj.eceasst.72.1008. URL `https://doi.org/10.14279/tuj.eceasst.72.1008`. (Cited on pages 4, 7, 52, 91, 139, 142, 204 and 209.)

[284] Gregor von Bochmann and Alexandre Petrenko. Protocol testing: Review of methods and relevance for software testing. In Thomas J. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, August 17-19, 1994*, pages 109–124. ACM, 1994. ISBN 0-89791-683-2. doi: 10.1145/186258.187153. URL `https://doi.org/10.1145/186258.187153`. (Cited on page 213.)

[285] Neil Walkinshaw and Gordon Fraser. Uncertainty-driven black-box test data generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 253–263. IEEE Computer Society, 2017. ISBN 978-1-5090-6031-3. doi: 10.1109/ICST.2017.30. URL `https://doi.org/10.1109/ICST.2017.30`. (Cited on page 212.)

[286] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_20. URL `https://doi.org/10.1007/978-3-642-05089-3_20`. (Cited on page 180.)

[287] Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris. Increasing functional coverage by inductive testing: A case study. In Alexandre Petrenko, Adenilso da Silva Simão, and

José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2010. ISBN 978-3-642-16572-6. doi: 10.1007/978-3-642-16573-3_10. URL `https://doi.org/10.1007/978-3-642-16573-3_10`. (Cited on page 207.)

[288] Felix Wallner. Benchmarking active automata learning configurations, 2019. Bachelor's thesis, Graz University of Technology. (Cited on pages 12, 55, 78, 79, 80 and 216.)

[289] Jingyi Wang, Jun Sun, and Shengchao Qin. Verifying complex systems probabilistically through learning, abstraction and refinement. *CoRR*, abs/1610.06371, 2016. URL `http://arxiv.org/abs/1610.06371`. (Cited on page 209.)

[290] Martin Weiglhofer and Franz Wotawa. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering, SE '08*, pages 286–291, Anaheim, CA, USA, 2008. ACTA Press. ISBN 978-0-88986-716-1. URL `http://dl.acm.org/citation.cfm?id=1722603.1722655`. (Cited on page 41.)

[291] Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. *International Journal of Software and Informatics*, 3(2-3):375–411, 2009. URL `http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=375&flag=1`. (Cited on page 213.)

[292] Elaine J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):641–655, 1983. doi: 10.1145/69575.357231. URL `https://doi.org/10.1145/69575.357231`. (Cited on page 29.)

[293] Tim A. C. Willemse. Heuristics for ioco-based test-based modelling. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers*, volume 4346 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006. ISBN 978-3-540-70951-0. doi: 10.1007/978-3-540-70952-7_9. URL `https://doi.org/10.1007/978-3-540-70952-7_9`. (Cited on page 209.)

[294] Håkan L. S. Younes. Probabilistic verification for "black-box" systems. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 253–265. Springer, 2005. ISBN 3-540-27231-3. doi: 10.1007/11513988_25. URL `https://doi.org/10.1007/11513988_25`. (Cited on page 209.)

[295] Haidi Yue and Joost-Pieter Katoen. Leader election in anonymous radio networks: Model checking energy consumption. In Khalid Al-Begain, Dieter Fiems, and William J. Knottenbelt, editors, *Analytical and Stochastic Modeling Techniques and Applications, 17th International Conference, ASMTA 2010, Cardiff, UK, June 14-16, 2010. Proceedings*, volume 6148 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2010. ISBN 978-3-642-13567-5. doi: 10.1007/978-3-642-13568-2_18. URL `https://doi.org/10.1007/978-3-642-13568-2_18`. (Cited on page 98.)

[296] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. *Model-Based Testing for Embedded Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011. ISBN 1439818452, 9781439818459. (Cited on page 2.)