Tomte: Bridging the Gap between Active Learning and Real-World Systems

Fides Aarts

Tomte: Bridging the Gap between Active Learning and Real-World Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. Th.L.M. Engelen,
volgens besluit van het college van decanen
in het openbaar te verdedigen op
maandag 27 oktober 2014 om 10:30 uur precies

door
Fides Dorothea Aarts
geboren op 10 november 1982
te Nijmegen

Promotor:

Prof. dr. F.W. Vaandrager

Manuscriptcommissie:

Prof. dr. J.H. Geuvers
Prof. dr. B. Jonsson (Uppsala Universitet, Zweden)
Prof. dr. B. Steffen (Technische Universität Dortmund, Duitsland)
Dr. M. Stoelinga (Universiteit Twente, Nederland)
Dr. ir. G.J. Tretmans

Het werk in dit proefschrift is uitgevoerd onder auspiciën van de onderzoeksschool IPA (Instituut voor Programmatuurkunde en Algoritmiek).

IPA Dissertation Series 2014-12

Dit onderzoek werd mede mogelijk gemaakt door het STW project 11763 (ITALIA, Integrating Testing And Learning of Interface Automata), `http://www.italia.cs.ru.nl/`, en het European Community's Seventh Framework Programme onder subsidienummer 214755 (QUASIMODO, Quantitative System Properties in Model-Driven Design), `http://www.quasimodo.aau.dk/`.

Omslag: Tobias Wählen
Druk: Gildeprint Drukkerijen

ISBN 978-90-9028499-6

Tomte: Bridging the Gap between Active Learning and Real-World Systems

DOCTORAL THESIS

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. Th.L.M. Engelen,
according to the decision of the Council of Deans
to be defended in public on
Monday, October 27, 2014 at 10:30 a.m.

by
Fides Dorothea Aarts
Born on November 10, 1982
in Nijmegen, the Netherlands

Supervisor:

Prof. dr. F.W. Vaandrager

Doctoral Thesis Committee:

Prof. dr. J.H. Geuvers
Prof. dr. B. Jonsson (Uppsala University, Sweden)
Prof. dr. B. Steffen (TU Dortmund University, Germany)
Dr. M. Stoelinga (University of Twente, the Netherlands)
Dr. ir. G.J. Tretmans

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 2014-12

This research was supported by the STW project 11763 (ITALIA, Integrating Testing And Learning of Interface Automata), `http://www.italia.cs.ru.nl/`, and the European Community's Seventh Framework Programme under grant agreement no. 214755 (QUASIMODO, Quantitative System Properties in Model-Driven Design), `http://www.quasimodo.aau.dk/`.

Cover: Tobias Wählen
Print: Gildeprint Drukkerijen

ISBN 978-90-9028499-6

Acknowledgements

With the writing of these acknowledgements, my PhD period comes to an end. During the past four and a half years, I had many wonderful people around me to whom I owe my gratitude for their help and support. To all of you: "Thank you!".

Nevertheless, there are some I would like to mention in particular. First and foremost, I would like to thank my supervisor Frits Vaandrager. Dear Frits, I am truly grateful for all the effort you made in finding funding to allow me to continue the research of my master's thesis. You guided my research, but you gave me the freedom to pursue my own interests. When I got stuck, you always found a solution and provided me with new ideas even though this area of research was also new to you. Your patience, profound knowledge, and experience have made a deep impression on me. I am very thankful for believing in me and recommending me for the Frye stipendium. You always know which step is the best one to take. I could not have had a better supervisor.

I owe a great debt of gratitude to everyone who supported me in the development of the Tomte tool. This includes Petur Olsen, who helped to set up the initial framework of the tool and to implement a first version. Faranak Heidarian assisted us in implementing the abstraction techniques. I am thankful to Gábor Angyal for his work on linking our Tomte tool with the CADP toolset. However, most of my thanks goes out to Harco Kuppens for the endless hours of programming we spent together on the tool – even during evenings, weekends, and holidays. Sometimes, it was frustrating when things did not work out, but your positive mood always made it an enjoyable time. Keep up your creativity and enthusiasm!

Apart from the Tomte tool, I have used several other tools for my experiments. I am grateful to Falk Howar from the TU Dortmund for his generous LearnLib support, and to Falk Howar and Bernhard Steffen for interesting discussions. Furthermore, I am thankful to Wojciech Mostowski for providing assistance with JMRTD.

This thesis is based on several papers, which are joint work with a number of coauthors: Faranak Heidarian, Falk Howar, Bengt Jonsson, Harco Kuppens, Petur Olsen, Erik Poll, Joeri de Ruiter, Julien Schmaltz, Jan Tretmans, Johan Uijen, Frits Vaandrager, and Sicco Verwer. I am grateful for the nice and fruitful collaborations. Without your contributions, this thesis would not have been the same. I also would like to thank the anonymous reviewers of my papers for their comments and suggestions.

I am happy about all the interest in my work and I appreciate the comments I received on earlier versions of this thesis. I thank the members of the reading committee, Herman Geuvers, Bengt Jonsson, Bernhard Steffen, Mariëlle Stoelinga, and Jan Tretmans for taking the time to carefully read my manuscript and for all the valuable remarks. Moreover, I thank Paul Fiterău-Broştean, Harco Kuppens, and Peter Schwabe for providing useful feedback on various draft chapters of this thesis. My thanks to Freek Verbeek for the LaTeX templates for this thesis.

My work was supported by the STW project ITALIA (Integrating Testing And Learning of Interface Automata) and the European project QUASIMODO. I appreciated the financial support as well as the occasional project meetings and discussions with all project members.

I also would like to thank my former Master's thesis supervisor Bengt Jonsson. He has introduced me to the world of active learning, which laid the cornerstone of this thesis. Bengt aroused my interest in scientific research and he stimulated me in pursuing this career. Without him I probably never would have started my PhD.

In the third year of my PhD, I got awarded the Frye grant, which I used for a research visit at Carnegie Mellon Silicon Valley. I am grateful to Christian von Essen, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Hoda Mehrpouyan, and Vishwanath Raman for their hospitality, the many enjoyable lunches with great food, and the amazing trips. This experience has been invaluable for me.

During my PhD I have attended the International Summer School Marktoberdorf and I have participated in many conferences and workshops. I am thankful for all the beautiful places I have seen, everything I have learned, and all the interesting conversations I have had with so many different people from around the world.

Special thanks go to my faithful companion Wenyun Quan. We started our PhD's exactly on the same day and have shared all the ups and downs of a PhD student's life. I am so happy to have met you and about all the wonderful moments we have experienced. I greatly appreciate your positive attitude and encouraging word in difficult times. I will see you at the finish line!

My working days have been made much more pleasant due to my colleagues. I thank everyone from the MBSD department, but also the rest of ICIS for the great atmosphere, coffee breaks, borrels, and organized events. Computer science is typically a male dominated field. Surprisingly, there were many female colleagues in our department. I would like to thank the MBSD girls, Faranak Heidarian, Georgeta Igna, Agnes Nakakawa, Wenyun Quan, Fiona Tulinayo, Marina Velikova, and Ilona Wilmont for their support, friendship, and the non-computer science related activities. I thank my office mates, Paul Fiterău-Broştean, Rick Smetsers, and Michele Volpato for the nice chats accompanied by maybe slightly too many bars of chocolate and other goodies. Aside from my office mates, I thank Harco Kuppens, Julien Schmaltz, and Jan Tretmans for regularly joining me for lunch at the university canteen. Despite the average quality meals, I will miss our fish and chips Friday at the Refter.

Also thanks to my friends from secondary school, HBO, university, in Germany, the Netherlands, or somewhere else for the necessary distraction from work. Many of you I have not seen on a regular basis, because I was too busy or away from

home. I am thankful for every email, phone call, and text message I received during this period.

An dieser Stelle möchte ich mich auch bei Tobias Eltern, Manfred und Bettina, bedanken für ihre liebe Unterstützung und aufmunternden Worte.

Tot slot wil ik me graag richten tot mijn familie. Ik wil mijn vader Carel en mijn zusje Shari bedanken voor hun aandacht en steun. Ook al begrepen ze niet altijd waar ik mee bezig was, waren ze altijd geïnteresseerd in hoe het met mij en mijn werk gaat.

Mein besonderer Dank gebührt meiner Mutter Birgit und meiner Oma Dorothea für ihre vielseitige Unterstützung, den notwendigen Rückhalt und ihr vorbehaltloses Vertrauen. Danke, dass ihr mir immer ermöglicht habt, dass ich mich zu 100 Prozent auf die Doktorarbeit konzentrieren kann. Ohne euch wäre ich nie soweit gekommen.

Den größten Dank möchte ich meinem Partner und besten Freund Tobias aussprechen. Ich weiß, dass es nicht immer einfach war mit mir in den vergangenen Jahren. Nichtsdestotrotz warst du immer für mich da, hast an mich geglaubt und warst mein Fels in der Brandung. Mit deinem Vertrauen und deiner Motivation hast du mich optimal unterstützt. Gleichzeitig hast du mich daran erinnert, was im Leben wirklich wichtig ist. Die "Diss" ist nun endlich fertig. Ich freue mich, den neuen Freiraum gemeinsam mit dir zu genießen.

Nijmegen, August 2014

Contents

List of Algorithms

Part I

Preamble

Chapter 1
Introduction

Printers, smart phones, ATMs, navigation systems and insulin pumps. They all have something in common: they penetrate many aspects of our society and everyday life. In addition, these devices can perform specific tasks, because they include one or more small computers. In contrast to a general-purpose computer, such as a personal computer (PC), they do not have a keyboard or display, nevertheless they contain numerous programs dedicated to handle particular functions. Such systems are called *embedded systems*. Embedded systems make products smarter. For example, think about a car that is equipped with an eye-tracking system to monitor the driver's fatigue. Embedded systems play an important role in solving societal challenges, such as sustainable energy, health of an aging population, national security, and safe and sustainable transportation. Another example is a dishwasher that automatically detects the soil level and adjusts the washing cycle accordingly to save power and water. Embedded systems have become vital to the quality of life and society.

Reliability of embedded systems plays a very important role. Many embedded systems are not monolithic systems built from scratch, but are constructed from many interacting components and sub-systems. Often, these larger systems are expected to run continuously for years. Even more critical, our lives may depend on embedded systems. Airbags in cars have to be triggered if the deployment threshold is met or exceeded, no matter if the occupants are belted, the road is rough or it is scorching heat. Therefore, embedded systems are usually tested more carefully than software for personal computers. Besides reliability, embedded systems must satisfy many more constraints. Applications are demanding greater functionality leading to larger and more complex systems. Also performance, energy efficiency, security and safety constraints need to be met.

In recent years, model-driven engineering (MDE) is attracting a lot of attention since it appears to be a software development methodology which can control the increasing complexity of embedded systems. In the MDE approach, the main objects of the software system being developed are represented at a higher level of abstraction using models. Specifications, behavior, functionality, development activities, and testing techniques of systems can all be described in terms of (graphical) models. These models can be used to generate implementations (semi)automatically or to promote communication between different

stakeholders like customers, product managers, designers, developers and users of the application domain. Moreover, models facilitate verification and validation, which refers to testing that the implementation complies with the requirements and the needs of the customer. Model-based techniques for verification and validation of different kinds of systems, including model checking [72, 43] and model-based testing [35] have witnessed drastic advances in the last decades, and several commercial tools that support model checking and/or model-based testing have become available (e.g., FDR2 [135], Reactis [133], Conformiq Designer [50], Sepp-Med MBTsuite [137], Axini Test Manager [16], and QuviQ [128]). Altogether, high-level models of system components speed up development cycles and reduce errors.

A key problem, however, is the construction of models that describe the intended behavior of the system components. Usually, this should be done during specification and design. In practice, often no models are created, because the manual construction is a very time-consuming and costly task [166]. Besides, most software projects are critical in terms of time. Therefore, it is essential to automatically generate such models, e.g. from an existing implementation. A potential approach is to use program analysis to construct models from source code [20, 76]. In many cases, however, access to source code is restricted due to the presence of library modules, third-party components, etc. Therefore, we consider an alternative approach by means of reverse engineering. We view the system as a black box and observe the external behavior by supplying inputs to the system and monitor how it responds to them. This allows us to make a significant step towards the automatic generation of models of real-world systems.

1.1 Active Automata Learning

Automatically constructing state transition models through observations is called *automata learning* (aka regular inference) and is a task which we apply frequently in our daily life. If we want to learn the behavior of a device or computer program, we often just press all available buttons and observe the resulting behavior. In this way, we construct a mental model of that device. We figure out in which global states the device can be and which state transitions and outputs occur in response to which input. This also explains why children know exactly how to turn on a TV or how an iPhone works without ever consulting a manual. A major challenge is to let computers perform similar learning tasks in a systematic manner for systems with large numbers of states.

The concept of automata learning has been studied in the literature for decades. During the last few years, it is gaining increasing attention since it has been demonstrated that automata-learning methods provide useful support for numerous software engineering tasks, especially related to testing and verification. An extensive list of applications in these areas is given in the related work section later in this chapter.

Automata learning is commonly divided into two categories: *passive learning* and *active learning*. In passive learning, a model is constructed from a given set of traces [159, 163, 53]. These traces usually originate from system logs, which have recorded the systems behavior over a specific period of time. A problem

with passive learning is that the quality of the model depends on the traces that have been observed. The larger the set of traces, the more likely that enough information is available to build an accurate model. However, for unobserved behavior it is difficult to draw a conclusion, which might lead to possible errors in the model. A solution to this problem is to request additional traces from the system in case of uncertainty. This is the strategy followed by the active learning approach. In active learning, a model is constructed while interacting with the system. Depending on the reaction of the system, the learning process can be steered and additional information from the system can be queried so that an accurate model can be constructed. In this thesis we consider an active learning approach.

Many active learning algorithms have been proposed since the 80s. Perhaps the most efficient and widely used one is the L^* algorithm of Angluin [14]. It was introduced in 1987 as an algorithm for learning *deterministic finite automata* (DFA, see, e.g., [149]). A DFA is a finite state machine that contains states, which are connected by means of transitions labeled with a symbol. Upon reading a symbol, a DFA jumps deterministically from a state to another by following the corresponding transition. A DFA accepts or rejects sequences of symbols, i.e., if a sequence of symbols leads to an accepting state, the sequence is said to be *accepted* by the DFA, otherwise *rejected*. Later, variations of the L^* algorithm were introduced [134, 96, 106, 19] differing in the data structure they use or in the way deviations between the inferred model and the real system are handled. Also inspired by the L^* algorithm, active learning has been extended to *Mealy machine* models [122, 107, 147]. A Mealy machine differs from a DFA in that its transitions are labeled with both an input symbol and the corresponding output symbol produced by the system, instead of just a single symbol. Mealy machines are particularly suitable to model reactive systems, which respond with an output rather than accepting or rejecting sequences of symbols. For example, a TV with a remote control is a reactive system, reacting to the user pressing the power button.

The active learning algorithm for Mealy machines has two participants: a *learner* and a *teacher*. The *teacher* knows a Mealy machine $\mathcal{M}$. The goal of the *learner* is to infer $\mathcal{M}$ by asking queries to the *teacher*. For this purpose, the *learner* knows the set of input symbols (the input alphabet) it can send to the *teacher*. The *learner* can ask two kinds of queries: output queries and equivalence queries. An output query poses the question "What is the output produced in response to input i?". Based on the answers from the *teacher*, the *learner* constructs a minimal hypothesis $\mathcal{H}$. The *learner* can test whether $\mathcal{H}$ is correct by asking an equivalence query to the *teacher*: "Is the hypothesized model $\mathcal{H}$ correct, i.e., equivalent to $\mathcal{M}$?". The *teacher* can reply with *yes* or provide a counterexample. If sufficiently many queries are asked, the learned automaton will be a model of the observed component.

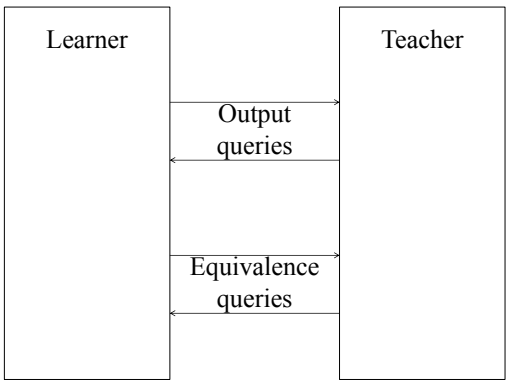


Figure 1.1: *Active learning*

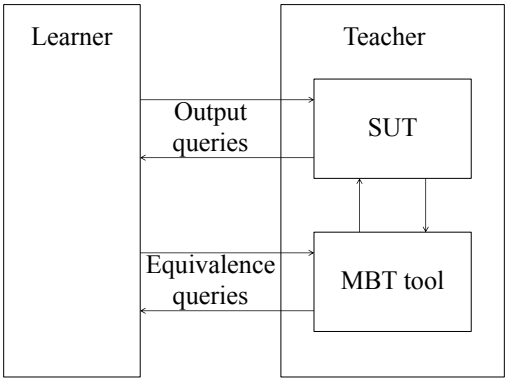


Figure 1.2: *Active learning of real-world systems*

Figure 1.2 illustrates how active learning can be used to infer models of real-world systems. A real-world system can be any (physical, embedded, software) reactive system to which we can apply inputs and whose outputs we may observe. In the remainder of this thesis, we often refer to these real-world systems as *system under test* (SUT) as they are the systems being inferred through observations and tests. The core of the *teacher* now is an SUT. The *learner* interacts directly with the SUT to infer a model by asking output queries. However, the SUT cannot answer equivalence queries, because it is a black box. Therefore, access to the internal states and transitions of the system model is not possible and, thus, it cannot be compared to the hypothesized model. A solution is to equip the *teacher* with a model-based testing (MBT) tool that approximates the equivalence queries using some model-based testing algorithm. For example, the MBT tool can generate long test sequences and compare whether the SUT and the hypothesis always produce the same output. Alternatively, experts or the system documentation can be consulted to answer equivalence queries. If no deviation has been found, we assume that the learned model is correct. Otherwise, a counterexample, which shows the difference between the SUT and the hypothesis, is returned to the *learner*. Hence, the task of the *learner* is to collect data by interacting with the SUT and to formulate hypotheses, and the task of the MBT tool is to test the correctness of these hypotheses. Checking correctness of a hypothesis is often the bottleneck in the application of active learning algorithms. The reason is that testing whether a black-box software system behaves according to a given model is hard [52]. Constructing informative test queries (i.e., output queries) is known as *test selection* and is one of the main problems dealt with in model-based software testing [100]. Several software testing methods and tools have been developed for this task in order to aid software development and maintenance, see, e.g., [25, 118, 138]. Although, these methods are able to approximate equivalence queries by (smartly) asking many output queries, this strategy may not be perfect: due to incomplete coverage, it may occur that the SUT passes the test for a hypothesis $\mathcal{H}$, even though it does not conform to $\mathcal{H}$.

Sometimes, however, we already have some knowledge about the system we want to learn. For example, there might exist a reference model that describes how the system should behave. To check whether the implemented system conforms to the reference model, we can apply active learning to generate a hypothesis of the implementation and then compare it to the reference model instead of using the MBT tool to approximate an equivalence query. This speeds up the learning process significantly, because the problems related to test completeness and selection can be circumvented. If the two models are equivalent, we have gained some confidence that the implementation conforms to the reference model and can further increase it by additional model-based testing. Otherwise, either the hypothesis is incorrect and has to be refined (if the counterexample trace produced by

the reference model is also a trace of the implementation) or a difference between the implementation and the reference model has been found. This method for improving conformance testing using active learning is based on the new notion of a so-called conformance oracle, which will be introduced in this thesis.

1.2 Bridging the Gap between Active Learning and Real-World Systems

Typically, state-of-the-art active learning algorithms only perform well if the input alphabet is small and the state machine has a moderate size. However, real-world systems are usually much larger, both in terms of the number of input symbols (due to data parameters in input messages) and in the number of states (due to the presence of state variables).

Example 1.1 Figure 1.3 depicts an *extended finite state machine* (EFSM) $\mathcal{M}$ in Mealy machine format of a login system that we will use as an example throughout this thesis. An EFSM may have extensions to its states and transitions. For example, its states may contain so called *state variables*, in which values may be stored. Its transitions may be extended with assignments to variables, and guards consisting of predicates which determine whether a transition can be taken or not. Actions in an EFSM may contain a list of parameters, each of which can assume different concrete values.

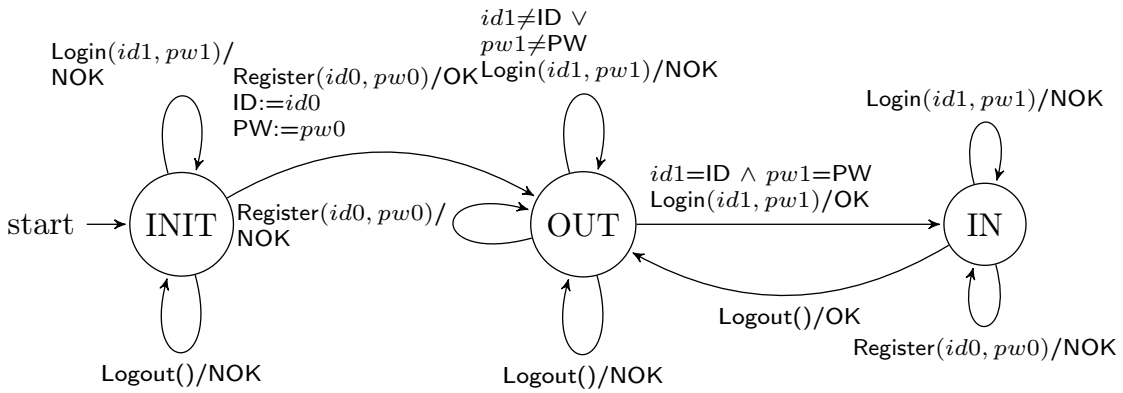


Figure 1.3: *Extended finite state machine in Mealy machine format of a login system*

The initial state of $\mathcal{M}$ is INIT. Here, a user may register with an ID and password. These values are stored in the corresponding state variables ID and PW. In the OUT state the user can login to the system by entering the values selected during the registration. Only in this case the guard statement is valid and the transition to the IN state can be taken. Being logged in, the user may log out again. The Register and Login input both contain two parameters, for which many different values can be entered. Alice may register with Register($Alice, Alice$), login to the system via Login($Alice, Alice$), but may forget her login credentials and try to login via Login($Alice, myPw$). Bob may create two accounts via Register($Bob, secret$) and Register($Bobby, 1234$). As one can see, there are infinitely many Register and Login input symbols with different combinations of ID and password. Thus, to

infer a good model of the login system using standard active learning techniques, the *learner* has to send many queries. □

The influence of data on control flow is also taken into account by several model-based test generation tools, such as ConformiQ Designer [85] and Spec Explorer [160, 67]. It is therefore important to extend inference techniques to handle messages containing data parameters with large domains.

Abstraction is the key when learning models of real-world systems. In most systems, sending the same input with different parameter values from the same state often leads to equivalent behavior. In the login system, for instance, invalid logins are handled in the same way. Hence, it would have been sufficient to send an output query with parameter values that cover a valid login and one with values that cover the invalid case. Our idea is to divide inputs with the same behavior into equivalence classes, which are then used as a new and reduced alphabet for the *learner*. The inference can be performed using the small alphabet, which is translated (in a history-dependent manner) to concrete messages via a so-called *mapper*, see Figure 1.4. In that way the communication with the SUT can be accomplished. Answers from the system are again translated back to abstract symbols of the reduced alphabet. By combining the abstract machine learned in this way with information from the mapper, it is possible to effectively learn an EFSM that is equivalent to the Mealy machine of the implementation. Roughly speaking, the *learner* is responsible for learning the global "control states" in which the system can be, and the transitions between those states, whereas the mapper records some relevant data in terms of state variables (typically computed from the parameters of previous input and output actions). The approach has been inspired by ideas from predicate abstraction [103], which has been successful for extending finite-state model checking to larger and even infinite state spaces. The introduction of a mapper component enables us to bridge the gap between active learning and real-world systems. However, the manual construction of a mapper typically is very time-consuming and requires specialized knowledge of the SUT. Therefore, it is desirable if such abstractions and mappers could be constructed fully automatically.

In this thesis, we discuss the entire process of constructing a suitable mapper component. We first formalize the fundamental notion of a mapper and the corresponding operations of abstraction and concretization. Then we show how a mapper can be constructed manually to learn models of realistic systems such as entities of the SIP and TCP protocol, the biometric passport, and the EMV protocol embedded in bank cards. After this, we present an algorithm to fully automatically infer models of a specific type of extended finite state machine. Automatically generating a mapper for any type of SUT is a mammoth task since the SUT may contain any type of operation, which is hard to infer in a black-box setting. Therefore, we to focus on a restricted class

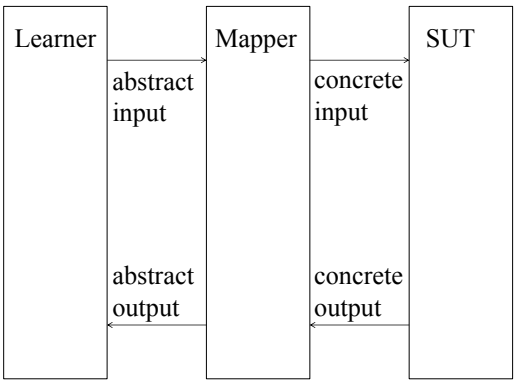


Figure 1.4: *Active learning using a mapper*

of systems, in which one can test for equality of data parameters, but no operations on data are allowed. We call this class of extended finite state machines *scalarset Mealy machines*, which also comprises the SIP and biometric passport case study conducted previously. Our approach uses counterexample-guided abstraction refinement (CEGAR), i.e., whenever the current abstraction is too coarse, it is refined automatically. We have implemented our algorithm in a tool called Tomte. It is named after the creature that shrank Nils Holgersson and after numerous adventures changed him back to his normal size again [98]. This is exactly what our Tomte tool is doing. It shrinks the input alphabet to be able to learn the system, and after numerous experiments, we can enlarge the abstract model again to a model of the original implementation by enriching it with information from the mapper component.

1.3 Related Work

This section positions the topic presented in this thesis with respect to other work in this area. After having discussed the early developments of automata learning, particularly of active learning, in the first part of this chapter, this section gives an overview of applications of automata learning and other used techniques as well as advances in bridging the gap between active learning and real-world systems.

Applications of automata learning Application areas include regression testing, replacing manual testing by model-based testing, producing models of standardized protocols, or analyzing whether an existing system is vulnerable to attacks. Regular inference techniques have been applied successfully to learn different types of complex systems such as control software embedded in printers and copiers [145], web-services [30], network protocols [51, 15, 49], and Java programs [168, 54, 108]. Moreover, they have been used for several tasks in test generation and verification. Related work on testing by means of automata learning techniques includes, for instance, regression testing of telecommunication systems [73, 86], integration testing [101, 71], security protocol testing [141], fuzz testing of protocol implementations [49], regression testing to create a specification and test suite [73, 86], and combining conformance testing and model checking [125, 70]. Furthermore, these methods have been used, e.g., to create models of environment constraints with respect to which a component should be verified [47], to perform model checking without access to source code or formal models [70, 125], for program analysis [13], and for formal specification and verification [47].

Learning different types of models There exists a wide variety of models for which learning algorithms have been developed: nondeterministic automata [170, 58, 165], probabilistic automata [42, 39], Petri-nets [158], timed automata [163, 68, 164, 69], I/O automata [11], Büchi automata [77], and Mealy machines [122, 107, 147, 139]. As mentioned in the previous section, the models that we consider in this thesis are scalarset Mealy machines. The notion of a scalarset data type originates from model checking, where it has been used for symmetry reduction [89]. Trying

to handle large or even infinite-state systems also motivated the recent work of Howar et al., who infer a canonical form of *register automata* [81, 38]. Similar to our scalarset Mealy machines, a register automaton is capable of expressing the influence of data on control flow. Moreover, register automata also operate on an infinite data domain, whose values can be assigned to registers (state variables) and compared for equality. However, register automata do not comprise parameterized output symbols. For this purpose, the authors developed an enhanced version of register automata, called *register Mealy machines* [80].

Other learning frameworks dealing with data Early work on inferring state machines with data can be found in [27]. Berg, Jonsson, and Raffelt present a modification of Angluin's algorithm to cope with models where the domain over which parameters range is large but finite. Unlike our approach, all parameters are booleans and parameters of possible output data are not considered. However, similar to our approach they partition input symbols into equivalence classes and refine theses classes whenever two symbols in the same class generate different reactions by the SUT. In this case the equivalence class is split by introducing a new guard, which is a conjunction over positive and negated parameter values. In later work, the authors extend regular inference to infinite-state state machines with input and output symbols from potentially infinite domains [28]. In a first phase they infer a Mealy machine model of the SUT using a small explicit data domain. In a second phase the Mealy machine is folded into a symbolic Mealy machine by capturing relations between data values.

Groz, Li, and Shahbaz [101, 140, 71] extend regular inference to Mealy machines with data values for use in integration testing, but use only a finite set of the data values in the obtained model. In particular, they do not infer internal state variables. In the work of Shu and Lee [141], parameters are essentially suppressed in order to obtain a finite subset of input symbols when learning the behavior of security protocol implementations. This subset can be extended in response to new information obtained in counterexamples. Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [109, 104]. They infer a control structure of possible sequence of method invocations. In addition, each invocation is annotated with a precondition on method parameters, possibly also correlated with accessible system variables. They use a passive learning approach where the model is inferred from a given sample of traces, forming the control structure by an extension of the k-tails algorithm, and using Daikon [36] to infer relations on method parameters. Their setup is that of passive learning: we use an active learning approach, where we assume that new queries may be supplied to the system; this is an added requirement but allows to generate a more informative sample by choosing the generated input. Furthermore, their work generates constraints that hold for the observed sample; they do not aim to infer functional relationships between input and output parameters, nor to infer how internal data variables of a component are managed.

Active learning algorithms for register automata and register Mealy machines have been implemented in the LearnLib tool [131, 114] and are presented in [81, 80]. Comparing their approach to our approach of inferring scalarset Mealy machines

shows that the main ideas are quite similar: both techniques consider a black-box setting, use the same underlying active Mealy machine learning algorithm, and apply a CEGAR-based procedure. What is different is the general architecture of the learning framework. Our structure is more modular by adding new functions as separate components like a mapper and a lookahead oracle. In contrast, Howar et al. perform all operations on the old *learner* component. They adapt the underlying data structure of the learning algorithm such that it stores all (relevant) data values. Comparing the details of both techniques reveals that there are more subtle differences. Since their approach is probably the one most similar to our approach, we dedicate a separate section on this in Chapter 7. A review of the development of active learning methods in the last decade dealing with data can be found in [146].

Abstraction is the key for scaling existing automata-learning methods to realistic applications. In order to learn models whose alphabet is large or infinite, Maler and Mens [105] define a symbolic representation, where transitions are labeled by predicates that are a subset of the alphabet. Their proposed method does not use any auxiliary variables, so that values of the input symbols can only be used within predicates of the current transition, but cannot be registered for future use.

Automatically generating precise interfaces for white-box components is a related area of research. For this purpose, the L^* algorithm is combined with predicate abstraction techniques in [12, 142] or symbolic execution in [66] to infer correct sequences of public method calls of an infinite-state component. For methods with parameters, method guards are used to define which actual values passed for the formal parameters correspond to an allowed method call [66]. In contrast to our work, parameter values are not stored in state variables so that these guards do not express the influence of data on control flow. Another difference is that the method guards can be retrieved from the component's source code using the underlying symbolic engine. Similar to our approach, the alphabet is then refined accordingly. Also the Σ^* technique by Botinčan and Babić [34] combines the L^* algorithm with symbolic execution and counterexample-guided abstraction refinement to learn symbolic models of stream filters. In this white-box approach symbolic execution is employed to extract predicates on input values and terms generating output values, which are then used to label the transitions of the so-called symbolic lookback transducers.

Other applications of mappers The problem of inferring state machines of real-world systems has gained a lot of interest in the past decade. Quite recently, mappers have become an important role in this field of research [93]. The idea of an intermediate component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning. Cho et al. [40] succeeded to infer models of realistic botnet command and control protocols by placing an emulator between botnet servers and the learning software, which concretizes the alphabet symbols into valid network messages and sends them to botnet servers. When responses are received, the emulator does the opposite — it abstracts the response messages into the output alphabet and passes them on to the learning software. In a similar way, Fiterău-Broştean, Janssen, and Vaandrager infer fragments of the TCP network protocol by using a mapper component, which

abstracts the large number of possible TCP packets into a limited number of abstract actions [91, 124].

Other uses of counterexample-guided abstraction refinement The idea to use CEGAR for learning state machines has also been explored by Howar at al. [83], who developed and implemented a CEGAR procedure for the special case in which the abstraction is static and does not depend on the execution history. Moreover, counterexample-guided abstraction refinement techniques have been widely used in the context of verification and model checking [44, 45, 32]. Several successful software verifiers are based on abstraction and CEGAR like the SLAM toolkit [21], the model checker BLAST [31], and the model checking tool SatAbs [46]. The approach by Clarke et al. [44, 45] is similar to our CEGAR-based construction of the mapper. Both approaches start with a coarse abstraction, but differ when a counterexample is found. In this case, we always refine the abstract model. In contrast, in the setting of Clarke et al. an actual counterexample corresponds to a program that does not satisfy a certain property and terminates the analysis, while a spurious counterexample is used to refine the abstraction.

More related work specific to the topics addressed in the following chapters can be found there.

1.4 Contributions of this Thesis

This thesis consists of four parts, which are divided into eight chapters. The contents of all chapters is based on eight publications, including two journal articles, five peer-reviewed conference papers, and one peer-reviewed workshop paper. The notation has been slightly adapted sometimes to make it uniform throughout the thesis. To avoid repetition, background information on Mealy machines and active learning required for all chapters has been combined in a separate chapter in Part I.

Part II is about active learning of *symbolic Mealy machines*. A symbolic Mealy machine is similar to an extended finite state machine in Mealy machine format, see Figure 1.3. It is also equipped with state variables to store data values, and the transitions may also contain assignments and guard statements. This part presents a general framework for learning models of systems with large or even infinite state spaces. In addition, it shows for particular realistic case studies how a mapper can be constructed manually. Part II contains the following major contributions:

1. We formalize the fundamental notion of a mapper and the corresponding operations of abstraction and concretization. Moreover, we show that a concrete model of a system can be constructed from its abstract model and the associated mapper.

2. Several case studies and experiments show the potential of the described technique to learn models of real-world systems. In this part we do not restrict the class of systems to be inferred, but consider different kinds of abstractions. In the case study about the session initiation protocol (SIP) a

mapper is defined manually to distinguish between different sessions being established. In contrast, in the transmission control protocol (TCP) case study a mapper is constructed with a number of predefined conditions to predict whether messages follow the protocol. When learning a model of the biometric passport, abstractions are needed to divide a number of files that contain passport data into equivalence classes, depending on when these files are readable. In the case study about inferring the EMV protocol, which is embedded in bank cards, the mapper is used to learn a counter. It keeps track of the value of the counter and can abstract when the counter has been incremented.

Part II is based on the following publications:

- Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. *Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction.* To appear in Formal Methods in System Design. 2014 [7]. This is the full version of the following paper:

- Fides Aarts, Bengt Jonsson, and Johan Uijen. *Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction.* In Proceedings ICTSS 2010, 22nd IFIP International Conference on Testing Software and Systems, Volume 6435 of Lecture Notes in Computer Science, pages 188–204, Natal, Brazil, November 8–12, 2010 [6].

- Fides Aarts, Julien Schmaltz, and Frits Vaandrager. *Inference and Abstraction of the Biometric Passport.* In Proceedings ISoLA 2010, 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Volume 6415 of Lecture Notes in Computer Science, pages 673–686, Crete, Greece, October 18–20, 2010 [10].

- Fides Aarts, Joeri de Ruiter, and Erik Poll. *Formal models of bank cards for free.* In Proceedings Software Testing, Verification and Validation Workshops (ICSTW 2013), 4th International Workshop on Security Testing (SECTEST), in association with the 6th International Conference on Software Testing, Verification and Validation (ICST), pages 461–468, Luxembourg City, Luxembourg, March 22, 2013 [1].

In Part III we focus on active learning of *scalarset Mealy machines*. A scalarset Mealy machine is a special case of a symbolic Mealy machine, in which parameter values may only be checked for equality against other values. This part presents techniques to automate active learning of scalarset Mealy machines. It contains the following major contributions:

3. We formalize and describe a procedure to construct the mapper automatically using a counterexample-guided abstraction refinement approach. Whenever the current abstraction is too coarse, it is refined automatically. Using Tomte, a prototype tool implementing our algorithm, we have succeeded to learn – fully automatically – models of several realistic software components, including the biometric passport and the SIP protocol.

4. We present the new notion of a lookahead oracle and provide algorithms to automatically detect the memorable values of an SUT. By enhancing our CEGAR-based approach with a lookahead extension (CEGAROLE), we can apply it to a great extent of scalarset Mealy machines.

5. We compare our approach to another algorithm for learning scalarset Mealy machines and highlight the differences and respective strengths.

Chapter 6 in Part III is based on the first publication mentioned below. Chapter 7 contains mainly new material and reports on joined work with Harco Kuppens and Frits Vaandrager. The comparison of different approaches is based on the second publication mentioned below.

- Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. *Automata Learning Through Counterexample-Guided Abstraction Refinement.* In Proceedings FM 2012, 18th International Symposium on Formal Methods, Volume 7436 of Lecture Notes in Computer Science, pages 10–27, Paris, France, August 27–31, 2012 [2].

- Fides Aarts, Falk Howar, Harco Kuppens, and Frits Vaandrager. *Algorithms for Inferring Register Automata - A comparison of existing approaches.* To appear in Proceedings ISoLA 2014, 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Corfu, Greece, October 8–11, 2014 [5].

Part IV demonstrates the diversity of application areas for which active learning can be used. It shows how conformance testing can be improved by combining automata learning and model-based testing, which is denoted by various terms like *learning-based testing* [112], *test-based modeling* [154], or *dynamic testing* [132, 129]. This part contains the following main contributions:

6. We show how active learning can be used to establish correctness of an implementation relative to a given reference implementation. For this purpose, we use a well-known industrial case study from the verification literature, the bounded retransmission protocol, and a unique combination of software tools for model construction (Uppaal), active learning (LearnLib, Tomte), model-based testing (JTorX, TorXakis) and verification (CADP, MRMC) that can be used for learning models of and revealing errors in implementations.

7. We present the new notion of a conformance oracle and demonstrate how conformance oracles can be used to speed up conformance checking.

Part IV is based on the following publications:

- Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. *Improving active Mealy machine learning for protocol conformance testing.* In Machine Learning, Volume 96, Issue 1–2, pages 189–224 , October 2013 [8]. This is the journal version of the following paper:

- Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. *Learning and Testing the Bounded Retransmission Protocol.* In

Proceedings ICGI 2012, 11th International Conference on Grammatical Inference, Volume 21 of JMLR Proceedings, pages 4–18, Washington, DC, USA, September 5–8, 2012 [9].

These papers have been selected, because they give a good overview of how active learning can be applied to real-world systems. They describe incrementally how a mapper component can be constructed – manually as well as automatically and show how our approach can be used to infer models of challenging real-world systems. In addition, the potential of active learning is demonstrated by using (improved) learning techniques to solve problems in new application areas like security and conformance testing. With respect to all papers, I was mainly involved in the development and discussion of new algorithms and ideas, the implementation of the Tomte tool and experiments, the execution of case studies, and writing the papers. Moreover, I contributed to the following papers, which are also related to active learning, but are not included in this dissertation:

- Fides Aarts and Frits Vaandrager. *Learning I/O Automata*. In Proceedings CONCUR 2010, 21th International Conference on Concurrency Theory, Volume 6269 of Lecture Notes in Computer Science, pages 71–85, Paris, France, August 31 – September 3, 2010 [11].
- Fides Aarts, Faranak Heidarian, and Frits Vaandrager. *A Theory of History Dependent Abstractions for Learning Interface Automata*. In Proceedings CONCUR 2012, 23rd International Conference on Concurrency Theory, Volume 7454 of Lecture Notes in Computer Science, pages 240–255, Newcastle upon Tyne, UK, September 3–8, 2012 [4].

Chapter 2
Active Learning of Mealy Machines

In this chapter, in order to fix notation and terminology, we recall the definition of a Mealy machine and the basic setup of active learning in Angluin-style.

2.1 Mealy Machines

We will use *Mealy machines* [110] to model real-world systems. In this thesis, real-world systems usually refer to reactive systems, which produce an output symbol in response to an input symbol, similar to the Mealy machine formalism.

Definition 2.1 (Mealy machine) A *(nondeterministic) Mealy machine* is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, where

- I, O, and Q are nonempty sets of *input symbols*, *output symbols*, and *states*, respectively,
- $q_0 \in Q$ is the *initial state*, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the *transition relation.*

We write $q \xrightarrow{i/o} q'$ if $(q, i, o, q') \in \rightarrow$, and $q \xrightarrow{i/o}$ if there exists a q' such that $q \xrightarrow{i/o} q'$. Mealy machines are assumed to be *input enabled* (or *completely specified*): for each state q and input i, there exists an output o such that $q \xrightarrow{i/o}$. We say that a Mealy machine is *finite* if the set Q of states and the set I of inputs are finite.

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in some state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $i \in I$. The machine then (nondeterministically) selects a transition $q \xrightarrow{i/o} q'$, produces output symbol o, and jumps to the new state q'.

Example 2.1 Figure 2.1 shows a Mealy machine based on an example in [120]. The set of inputs is $I = \{0, 1\}$, the set of outputs is $O = \{0, 1, 2\}$, and the set of states is given by $Q = \{q_0, q_1, q_2\}$, where q_0 is the initial state marked with an extra circle. The transition relation is defined by Table 2.1: □

	Input			
	0		1	
Source state	Output	Target state	Output	Target state
q_0	0	q_0	1	q_1
q_1	2	q_2	0	q_0
q_2	1	q_1	2	q_2

Table 2.1: *Transition relation for the Mealy machine in Figure 2.1*

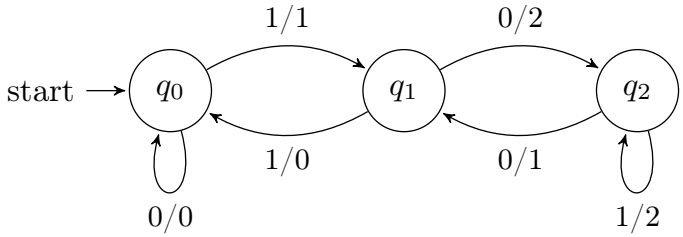


Figure 2.1: *A simple Mealy machine*

Behavior of Mealy Machines The transition relation $\rightarrow$ is extended to finite sequences by defining $\stackrel{u/s}{\Rightarrow}$ to be the least relation that satisfies, for $q, q', q'' \in Q$, $u \in I^*$, $s \in O^*$, $i \in I$, and $o \in O$,

- $q \stackrel{\epsilon/\epsilon}{\Rightarrow} q$, and
- if $q \stackrel{i/o}{\longrightarrow} q'$ and $q' \stackrel{u/s}{\Rightarrow} q''$ then $q \stackrel{i\,u/o\,s}{\Rightarrow} q''$.

Here we use ϵ to denote the empty sequence. We write $|s|$ to denote the length of a sequence s. Observe that $q \stackrel{u/s}{\Rightarrow} q'$ implies $|u| = |s|$. A state $q \in Q$ is called *reachable* if $q_0 \stackrel{u/s}{\Rightarrow} q$, for some u and s.

Example 2.2 For example, the machine in Figure 2.1 maps 010 to 012 (path: $q_0 \stackrel{0/0}{\longrightarrow} q_0, q_0 \stackrel{1/1}{\longrightarrow} q_1, q_1 \stackrel{0/2}{\longrightarrow} q_2$), 1101 is mapped to 1001, and 1110 to 1012. As one can see it computes residues modulo 3 for a binary input (most significant bit first) number. Given an input word $i_1 \ldots i_n$, the jth output of the machine corresponds to $i_1 \ldots i_j$ modulo 3. For input 010 mentioned before it computes 0 mod 3 = 0, 1 mod 3 = 1, 2 mod 3 = 2, for input 1101 it computes 1 mod 3 = 1, 3 mod 3 = 0, 6 mod 3 = 0, 13 mod 3 = 1, and for input 1110 it computes 1 mod 3 = 1, 3 mod 3 = 0, 7 mod 3 = 1, 14 mod 3 = 2. □

An *observation* over input symbols I and output symbols O is a pair $(u, s) \in I^* \times O^*$ such that sequences u and s have the same length. For $q \in Q$, we define $obs_{\mathcal{M}}(q)$, the set of observations of $\mathcal{M}$ from state q, by

$$obs_{\mathcal{M}}(q) \quad = \quad \{(u, s) \in I^* \times O^* \mid \exists q' : q \stackrel{u/s}{\Rightarrow} q'\}.$$

We write $obs_{\mathcal{M}}$ as a shorthand for $obs_{\mathcal{M}}(q_0)$. Note that, since Mealy machines are input enabled, $obs_{\mathcal{M}}(q)$ contains at least one pair (u, s), for each input sequence

$u \in I^*$. We call $\mathcal{M}$ *behavior deterministic* if $obs_{\mathcal{M}}$ contains exactly one pair (u, s), for each $u \in I^*$. (In the literature on transducers the term *single-valued* is used instead [161].)

Two states $q, q' \in Q$ are *observation equivalent*, denoted $q \approx q'$, if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same sets of input symbols are *observation equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$. We say that $\mathcal{M}_1$ *implements* $\mathcal{M}_2$, notation $\mathcal{M}_1 \leq \mathcal{M}_2$, if $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same sets of input symbols and $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2}$.

The following lemma easily follows from the definitions.

Lemma 2.1 Suppose $\mathcal{M}_1 \leq \mathcal{M}_2$ and $\mathcal{M}_2$ is behavior deterministic. Then $\mathcal{M}_1 \approx \mathcal{M}_2$.

Proof. It suffices to prove $obs_{\mathcal{M}_2} \subseteq obs_{\mathcal{M}_1}$. Assume $(u, s) \in obs_{\mathcal{M}_2}$. Since $\mathcal{M}_1$ is input enabled, there exists an s' such that $(u, s') \in obs_{\mathcal{M}_1}$. Since $\mathcal{M}_1 \leq \mathcal{M}_2$, $(u, s') \in obs_{\mathcal{M}_2}$. Because $\mathcal{M}_2$ is behavior deterministic, $s = s'$. Thus $(u, s) \in obs_{\mathcal{M}_1}$, as required. □

We say that a Mealy machine is *finitary* if it is observation equivalent to a finite Mealy machine.

Example 2.3 Trivially, each finite Mealy machine is finitary. An example of a finitary Mealy machine that is not finite is a Mealy machine with as states the set $\mathbb{N}$ of natural numbers, initial state 0, a single input i and a single output o, and transitions $n \xrightarrow{i/o} n+1$. This Mealy machine, which records in its state the number of inputs that has occurred, is observation equivalent to a Mealy machine with a single state q_0 and a single transition $q_0 \xrightarrow{i/o} q_0$. □

Deterministic Mealy Machines A Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, is *deterministic* if for each state q and input symbol i there is exactly one output symbol o and exactly one state q' such that $q \xrightarrow{i/o} q'$. A deterministic Mealy machine $\mathcal{M}$ can equivalently be represented as a structure $\langle I, O, Q, q_0, \delta, \lambda \rangle$, where update function $\delta : Q \times I \to Q$ and output function $\lambda : Q \times I \to O$ are defined by:

$$q \xrightarrow{i/o} q' \quad \Rightarrow \quad \delta(q, i) = q' \wedge \lambda(q, i) = o.$$

Update function δ is extended to a function from $Q \times I^* \to Q$ by the following classical recurrence relations:

$$\begin{aligned} \delta(q, \epsilon) &= q, \\ \delta(q, i\ u) &= \delta(\delta(q, i), u). \end{aligned}$$

Similarly, output function λ is extended to a function from $Q \times I^* \to O^*$ by

$$\begin{aligned} \lambda(q, \epsilon) &= \epsilon, \\ \lambda(q, i\ u) &= \lambda(q, i)\ \lambda(\delta(q, i), u). \end{aligned}$$

Example 2.4 It is easy to see that a deterministic Mealy machine is behavior deterministic. Figure 2.2 gives an example of a behavior deterministic Mealy machine that is not deterministic. □

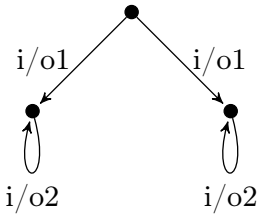


Figure 2.2: *Mealy machine that is behavior deterministic but not deterministic*

2.2 Active Learning

In order to learn Mealy machines, we define a slight variation of the active learning setting of Angluin [14]. The basic setup for active learning is illustrated in Figure 2.3, similar to the one presented in Figure 1.2.

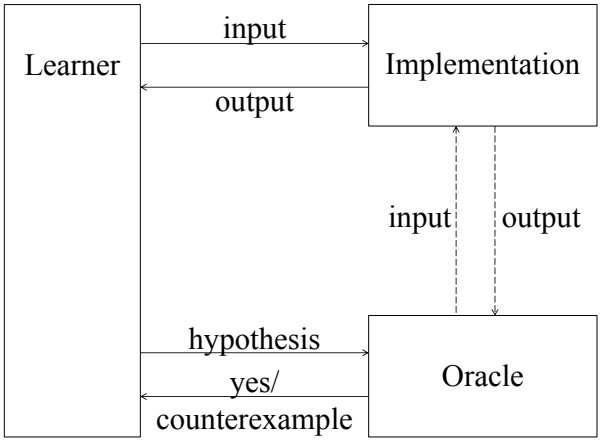


Figure 2.3: *Basic setting for active learning of real-world systems. A* learner *starts by providing inputs (output queries) to an implementation of the system (SUT), the implementation produces the corresponding outputs. By intelligently choosing these inputs, the* learner *is able to form a hypothesis model of the implementation's internal behavior. This model is provided to an oracle (equivalence/inclusion queries). The oracle then tests whether this model correctly describes the inner workings of the implementation. For this purpose, the oracle (e.g. a MBT tool) may communicate with the implementation, see the dashes lines, but it may also use other sources of knowledge. If the hypothesis is correct, the oracle returns* yes*. Otherwise, it returns a counterexample demonstrating this incorrectness.*

Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a behavior deterministic Mealy machine. An *implementation* of $\mathcal{M}$ is a device that maintains the current state of $\mathcal{M}$, which at the beginning equals q_0. An implementation of $\mathcal{M}$ accepts inputs in I, called *output queries*, as well as a special reset input. Upon receiving query $i \in I$, the implementation picks a transition $q \xrightarrow{i/o} q'$, where q is its current state, generates output $o \in O$, and updates its current state to q'. Upon receiving a reset, the implementation resets its current state to q_0. In contrast to Angluin's setting, where the *learner* asks whether an input string is a member of a language, we ask for the outputs produced. Therefore, the term *output queries* is more appropriate than *membership queries*.

An *oracle* or *teacher* for $\mathcal{M}$ is a device which accepts an *inclusion query* or *hypothesis* $\mathcal{H}$ as input, where $\mathcal{H}$ is a Mealy machine with inputs I. Upon receiving a hypothesis $\mathcal{H}$, an oracle for $\mathcal{M}$ will produce output *yes* if the hypothesis is correct, that is, $\mathcal{M} \leq \mathcal{H}$, or else output a *counterexample*, which is an observation $(u, s) \in obs_{\mathcal{M}} - obs_{\mathcal{H}}$. The combination of an implementation of $\mathcal{M}$ and an oracle for $\mathcal{M}$ corresponds to what Angluin [14] calls a *teacher* for $\mathcal{M}$.

A *learner* for I is a device that may send inputs in $I \cup \{\mathsf{reset}\}$ to an implementation of $\mathcal{M}$, and Mealy machines $\mathcal{H}$ over I to an oracle for $\mathcal{M}$. The task of the *learner* is to learn a correct hypothesis in a finite number of steps, by observing the outputs generated by the implementation and the oracle in response to the queries.

Note that *inclusion queries* are slightly more general than the *equivalence queries* used by Angluin [14] and Niese [122]. However, if $\mathcal{M} \leq \mathcal{H}$ and moreover $\mathcal{H}$ is behavior deterministic then $\mathcal{M} \approx \mathcal{H}$ by Lemma 2.1. Hence, a deterministic hypothesis is correct in our setting iff it is correct in the settings of Angluin and Niese. For case studies where we consider deterministic SUTs, we use both terms synonymously when discussing experimental results. The reason for our generalization will be discussed in Section 3.1.4.

The typical behavior of a *learner* is to start by asking sequences of output queries (alternated with resets) until a "stable" hypothesis $\mathcal{H}$ can be built from the answers. After that an inclusion query is made to find out whether $\mathcal{H}$ is correct. If the answer is *yes* then the *learner* has succeeded. Otherwise the returned counterexample is used to perform subsequent output queries until converging to a new hypothesized automaton, which is supplied in an inclusion query, etc.

Below we give a simplified presentation of the L^* learning algorithm as it has been implemented, for instance, in the LearnLib tool [131, 114]. The actual algorithm of LearnLib contains several optimizations that are explained in [147, 114].

In order to organize the collected observations, the *learner* maintains an observation table $\mathcal{O}$. An observation table is a tuple $\mathcal{O} = (S, E, T)$ consisting of a nonempty finite prefix-closed set S of strings, a nonempty finite suffix-closed set E of strings and a function T, which maps $(S \cup (S \cdot I)) \cdot E$ to a symbol from the output alphabet O, where $\cdot$ denotes the concatenation of strings. Intuitively, S characterizes the states in the automaton, E is used to distinguish states by their future behavior, and T corresponds to a function that returns the last output symbol produced in response to a sequence of output queries.

An observation table can be viewed as a table with the elements of $S \cup (S \cdot I)$ representing the rows and the elements of E labeling the columns, see Table 2.2. An entry in the table identifies the last symbol of the output string that is produced after the sequence of input symbols, defined by its row and column accordingly, is executed.

	E
S	O
$S \cdot I$	O

Table 2.2: *Example of an observation table*

To construct a Mealy machine from the observation table, it must fulfill two criteria. It has to be *closed* and *consistent*. We say that an observation table is

- *closed* if all transitions lead to already established states, i.e. if for each

$w' \cdot a \in S \cdot I$ there exists a string $w \in S$ that has the same answer to the corresponding output query, thus $row(w' \cdot a) = row(w)$, and

- *consistent* if identically characterized states show the same future behavior, i.e. if whenever $w_1, w_2 \in S$ are such that $row(w_1) = row(w_2)$, then for all $a \in I$ we have $row(w_1 \cdot a) = row(w_2 \cdot a)$.

As described before, the *learner* maintains the observation table $\mathcal{O} = (S, E, T)$, where initially S contains the single element $\{\varepsilon\}$ and E is initialized with the whole set of input symbols I. The *learner* starts by asking output queries of form $((S \cup (S \cdot I)) \cdot E)$ to fill the fields in the table. Each entry in the table is filled with an element of the output alphabet O representing the last symbol of the answer. After this, it is checked whether the given observation table fulfils the conditions of closedness and consistency.

If $\mathcal{O}$ is not closed, then the *learner* finds a $w' \in S$ and $a \in I$ such that $row(w' \cdot a) \neq row(w)$ for all $w \in S$. In this case, the *learner* adds $w' \cdot a$ to S and asks output queries for all the strings of the form $w' \cdot a \cdot b \cdot e$, where $e \in E$, $b \in I$ and $w' \cdot a \cdot b$ corresponds to a row that has to be added to the lower part of the table.

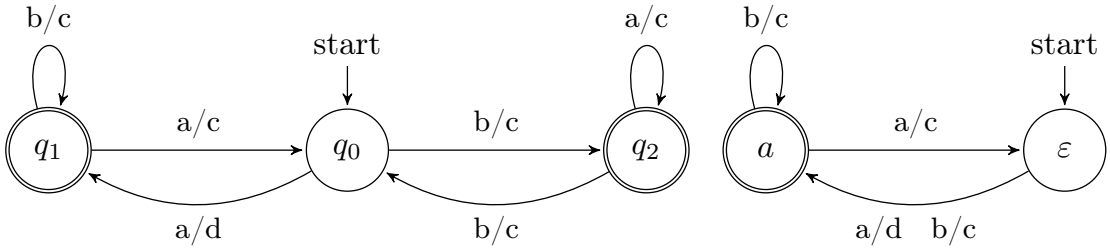
If $\mathcal{O}$ is not consistent, then the *learner* finds two strings $w_1, w_2 \in S$, $e \in E$ and $a \in I$ such that $row(w_1) = row(w_2)$, but $T(w_1 \cdot a \cdot e) \neq T(w_2 \cdot a \cdot e)$. Then the *learner* adds the string $a \cdot e$ to E and asks output queries in order to fill the missing fields in the new column.

When $\mathcal{O}$ is closed and consistent, it is possible to construct the corresponding deterministic Mealy machine $\mathcal{H} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ as follows:

- $Q = \{row(w) | w \in S\}$ is the set of *distinct* rows,
- $q_0 = row(\varepsilon)$ is the initial state,
- $\delta(row(w), a) = row(w \cdot a)$,
- $\lambda(row(w), a) = T(w \cdot a)$.

The *learner* creates the hypothesized automaton $\mathcal{H}$ and asks an inclusion query to the *teacher*. If the *teacher* replies with *yes*, then the algorithm terminates with output $\mathcal{H}$. Otherwise a counterexample (u,s) is returned, where u, including all its prefixes u', is added to S. Then the *learner* asks output queries for the missing entries.

Example 2.5 Let us consider an example Mealy machine $\mathcal{M}$ based on Sipser [143], which we want to infer. The automaton $\mathcal{M}$ is depicted in Figure 2.4(a). We start by asking output queries for a, b, aa, ab, ba, and bb. The answers are filled in the initial observation table $\mathcal{O}_1$ shown in Table 2.3(a), where $S = \{\varepsilon\}$ and $E = \{a, b\}$. This table is not closed since $row(a) \neq row(\varepsilon)$ and $row(b) \neq row(\varepsilon)$. As they are equal, it is sufficient to add one of them to S. Thus, a is moved to S and $\mathcal{O}$ is extended by asking output queries for aaa, aab, aba, and abb, see $\mathcal{O}_2$ shown in Table 2.3(b). Now the table is both closed and consistent. The *learner* can make a first guess by constructing the hypothesized automaton $\mathcal{H}$ shown in Figure 2.4(b) and asking an inclusion query to the *teacher*. The *teacher* rejects $\mathcal{H}$ and replies with a counterexample - assume (bba,ccd), where $\mathcal{H}$ produces ccc. To



(a) An example Mealy machine $\mathcal{M}$

(b) A hypothesized Mealy machine $\mathcal{H}$

Figure 2.4: *A Mealy machine to be inferred and a hypothesized Mealy machine*

process the counterexample, we add bba and all its prefixes (b and bb) to S. S is now $\{\varepsilon, a, b, bb, bba\}$ and we ask output queries for all $((S \cup (S \cdot I)) \cdot E)$. The newly constructed observation table $\mathcal{O}_3$ is depicted in Table 2.3(c). This observation table is no longer consistent since $row(a) = row(b)$ but $row(aa) \neq row(ba)$. So we add aa to E and ask output queries to fill the new column. This results in observation table $\mathcal{O}_4$, which is shown in Table 2.3(d). This table is closed and consistent, so that we can make a second guess and ask an inclusion query to the *teacher*. The *teacher* replies with *yes*, i.e. the hypothesized automaton is equal to $\mathcal{M}$ and the algorithm terminates. Note that in Table 2.3(d) $row(\varepsilon) = row(bb)$ and $row(a) = row(bba)$. Because Q is the set of *distinct* rows, the hypothesized Mealy machine $\mathcal{H}$ merges these states and as a result contains three states equivalent to the states of $\mathcal{M}$.

(a) $\mathcal{O}_1$

$\mathcal{O}_1$	a	b
ε	d	c
a	c	c
b	c	c

(b) $\mathcal{O}_2$

$\mathcal{O}_2$	a	b
ε	d	c
a	c	c
b	c	c
aa	d	c
ab	c	c

(c) $\mathcal{O}_3$

$\mathcal{O}_3$	a	b
ε	d	c
a	c	c
b	c	c
bb	d	c
bba	c	c
aa	d	c
ab	c	c
ba	c	c
bbb	c	c
$bbaa$	d	c
$bbab$	c	c

(d) $\mathcal{O}_4$

$\mathcal{O}_3$	a	b	aa
ε	d	c	c
a	c	c	d
b	c	c	c
bb	d	c	c
bba	c	c	d
aa	d	c	c
ab	c	c	d
ba	c	c	c
bbb	c	c	c
$bbaa$	d	c	c
$bbab$	c	c	d

Table 2.3: *Observation tables*

□

For finitary, behavior deterministic Mealy machines the above problem is well understood. The L^* algorithm has been adapted to Mealy machines by Niese [122], which again has been optimized by Steffen et al. in the L^*_M algorithm [147],

implemented in the LearnLib tool [131, 114]. LearnLib is the winner of the 2010 Zulu competition on regular inference [82, 48] and, currently, is able to learn state machines with at most 10,000 states. Both learning algorithms generate deterministic hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with a performed set of output queries. In theory, we have defined an output query as a single input symbol that is sent to an implementation. For readability, however, and to keep numbers as small as possible, we have decided to count sequences of inputs (separated by resets) when learning or verifying a hypothesis. Especially if performing long test traces, counting single inputs results in huge numbers that are difficult to compare. Since in practice there is no oracle that can answer equivalence or inclusion queries, LearnLib "approximates" such an oracle by generating long test sequences using standard methods like state cover, transition cover, or the W-method. These test sequences are then applied to the implementation to check if the produced output agrees with the output predicted by the hypothesis. In this thesis, we use LearnLib as the basic active learning tool, but there exist also other libraries like libalf [33] that implement active learning algorithms.

2.3 Inference Using Subalphabets

If an implementation $\mathcal{M}$ has a large set of input symbols, learning a model for $\mathcal{M}$ may become difficult, in particular the construction of a good testing oracle. In Section 3.1, we will explore the use of abstractions to reduce the size of the input alphabet. A very simple orthogonal strategy, which has been successfully used in [144], is to first learn a model for a small subset of the input alphabet, and to extend this model in a stepwise fashion by enlarging the subset of input symbols considered.

In this approach, rather than learning a model for a Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, we learn a model for the Mealy machine $\mathcal{M} \downarrow J$, for some subset of input symbols $J \subseteq I$. Here $\mathcal{M} \downarrow J$, the *restriction* of $\mathcal{M}$ to subalphabet J, is the Mealy machine $\langle J, O, Q, q_0, \rightarrow' \rangle$, where $\rightarrow' = \{(q, i, o, q') \in \rightarrow \mid i \in J\}$.

Example 2.6 The restriction operator is illustrated by a simple example of a two-place buffer in Figure 2.5.

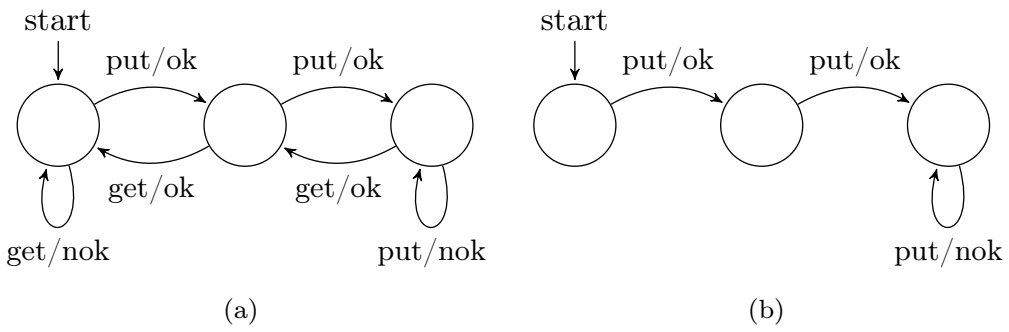


Figure 2.5: *Mealy machine $\mathcal{M}_1$ for a two-place buffer (a) and the restriction $\mathcal{M}_1 \downarrow \{\textbf{put}\}$ (b).*

□

The next lemma, which follows immediately from the definitions, characterizes the set of traces of the restriction.

Lemma 2.2 Let $\mathcal{M}$ be a Mealy machine with input alphabet I and let $J \subseteq I$. Then $obs_{\mathcal{M}\downarrow J} = \{(u, s) \in obs_{\mathcal{M}} \mid u \in J^*\}$.

The adjoint *extension* operator enlarges the set of inputs of a Mealy machine. Whenever an input arrives that is not in the original input alphabet, the extension moves to a "chaos" state χ in which any behavior is possible. Formally, if $J \supseteq I$, then $\mathcal{M} \uparrow J$, the *extension* of $\mathcal{M}$ to input alphabet J, is the Mealy machine $\langle J, O, Q \cup \{\chi\}, q_0, \rightarrow' \rangle$, where $\chi \notin Q$ is a fresh state and

$$\rightarrow' \;=\; \rightarrow \cup \{(q, i, o, \chi) \mid q \in Q \wedge i \in J - I \wedge o \in O\} \cup \{(\chi, i, o, \chi) \mid i \in J \wedge o \in O\}.$$

Example 2.7 The extension operator is illustrated in Figure 2.6. Here a transition with output **any** abbreviates two transitions with outputs **ok** and **nok**, respectively.

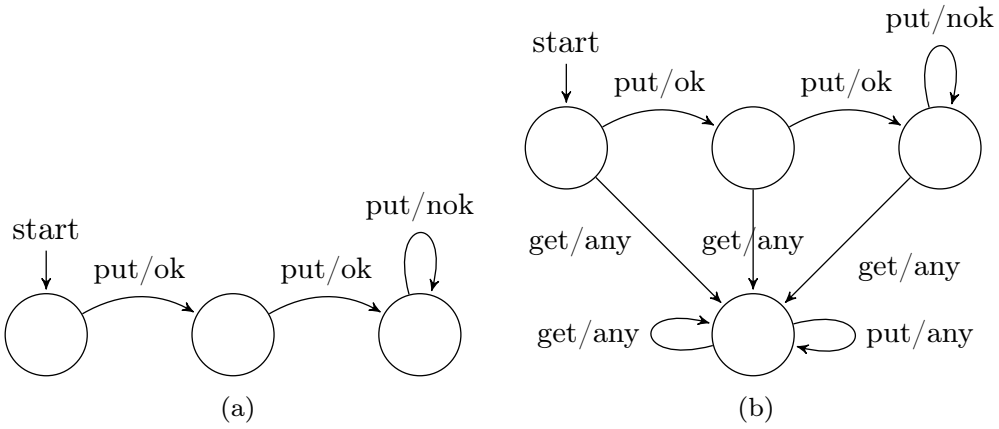


Figure 2.6: *Mealy machine $\mathcal{M}_2$ with inputs* {**put**} *(a) and the extension $\mathcal{M}_2 \uparrow$* {**put**, **get**} *(b).*

□

The next lemma, which follows immediately from the definitions, characterizes the set of traces of the extension.

Lemma 2.3 Let $\mathcal{M}$ be a Mealy machine with input alphabet I and let $J \supseteq I$. Then

$$\begin{aligned} obs_{\mathcal{M}\uparrow J} \;=\; & obs_{\mathcal{M}} \cup \\ & \{(u_1\ i\ u_2, s_1\ s_2) \in J^* \times O^* \mid (u_1, s_1) \in obs_{\mathcal{M}} \wedge i \in J - I \wedge \\ & |u_2| = |s_2| - 1\}. \end{aligned}$$

Lemma's 2.2 and 2.3 imply that the extension and restriction operators are monotone. In combination with the following result, it follows that the restriction and extension operators together constitute a Galois connection.

Lemma 2.4 Let, for $i = 1, 2$, $\mathcal{M}_i = \langle I_i, O_i, Q_i, q_i^0, \rightarrow_i \rangle$ be Mealy machines with $I_1 \supseteq I_2$ and $O_1 = O_2$. Then $\mathcal{M}_1 \downarrow I_2 \leq \mathcal{M}_2$ iff $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow I_1$.

Proof. Suppose $\mathcal{M}_1 \downarrow I_2 \leq \mathcal{M}_2$. We must prove $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2 \uparrow I_1}$. Suppose $(u, s) \in obs_{\mathcal{M}_1}$. We consider two cases.

1. $u \in I_2^*$. Then, by Lemma 2.2, $(u, s) \in obs_{\mathcal{M}_1 \downarrow I_2}$. By assumption, $(u, s) \in obs_{\mathcal{M}_2}$. By Lemma 2.3, $(u, s) \in obs_{\mathcal{M}_2 \uparrow I_1}$, as required.

2. u is of the form $u_2 i u_1$ with $u_2 \in I_2^*$, $i \in I_1 - I_2$ and $u_1 \in I_1^*$. Let s_2 be the prefix of s with length equal to u_2. Since the set of observations of a Mealy machine is prefix closed, $(u_2, s_2) \in obs_{\mathcal{M}_1}$. By Lemma 2.2, $(u_2, s_2) \in obs_{\mathcal{M}_1 \downarrow I_2}$. Hence, by the assumption, $(u, s) \in obs_{\mathcal{M}_2}$. By Lemma 2.3, $(u, s) \in obs_{\mathcal{M}_2 \uparrow I_1}$, as required.

In order to prove the converse implication, suppose $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow I_1$. We must prove $obs_{\mathcal{M}_1 \downarrow I_2} \subseteq obs_{\mathcal{M}_2}$. Suppose $(u, s) \in obs_{\mathcal{M}_1 \downarrow I_2}$. Then, by Lemma 2.2, $u \in I_2^*$ and $(u, s) \in obs_{\mathcal{M}_1}$. Thus, by the assumption, $(u, s) \in obs_{\mathcal{M}_2 \uparrow I_1}$. Using $u \in I_2^*$ it follows, by Lemma 2.3, that $(u, s) \in obs_{\mathcal{M}_2}$, as required. □

The Mealy machines of Figures 2.5 and 2.6 may be used to illustrate Lemma 2.4. Clearly $\mathcal{M}_1 \downarrow \{\mathbf{put}\} \leq \mathcal{M}_2$. The reader may check that $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow \{\mathbf{put}, \mathbf{get}\}$.

Part II

Active Learning of Symbolic Mealy Machines

Chapter 3
Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction

In this chapter, we present a general framework for generating models of protocol components with large or infinite structured message alphabets and state spaces. We use an externally supplied abstraction layer, which translates between a large or infinite message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

We describe how to construct a suitable abstraction from knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component. We have implemented our techniques by connecting the LearnLib tool for regular inference with an implementation of SIP in ns-2 and an implementation of TCP in Windows 8, and generated models of SIP and TCP components.

Contribution The main contribution of this chapter is a formalization of the fundamental notion of a mapper component and associated operations of abstraction and concretization, and some results (in particular Theorems 3.1 and 3.3) that allow us to construct a concrete model of a system from abstract model and a mapper. Technically, a mapper is a deterministic Mealy machine with additional structure, and as such a specific type of transducer (not necessarily finite state). However, the operations that we define for mappers (abstraction and concretization) are new and different from the operations usually considered for transducers such as union, Kleene closure, and composition [116]. Two case studies and experiments show the potential of the described technique to learn interfaces of real protocol implementations.

Organization Our new abstraction technique is presented in Section 3.1. Section 3.2 discusses the symbolic representation of Mealy machines and mappers.

Section 3.3 describes how mappers can be constructed in a systematic way. The application to SIP and TCP is reported in Section 3.4. Section 3.5 contains conclusions and directions for future work. Appendices 3.A, 3.B and 3.C display the (abstract) models that we learned for the SIP and TCP protocols.

3.1 Inference using Abstraction

Existing implementations of inference algorithms only proved effective when applied to machines with small alphabets (sets of input and output symbols). Practical systems, however, typically have large alphabets, e.g. inputs and outputs with data parameters of type integer or string. In order to infer large- or infinite-state Mealy machines, we adapt ideas from predicate abstraction [103, 45], which have been successful for extending finite-state model checking to large and infinite state spaces. The main idea is to divide the concrete input domain into a small number of abstract equivalence classes in a history-dependent manner.

Example 3.1 (Component of a simple communication protocol) Consider a Mealy machine $\mathcal{M}_{COM}$ that models a component of a simple communication protocol. The component accepts request messages, which are modeled as inputs of $\mathcal{M}_{COM}$, and generates OK/NOK reply messages, which correspond to outputs of $\mathcal{M}_{COM}$. The set of inputs is $I = \{REQ(id, sn) \mid id, sn \in \mathbb{N}\}$, where parameter id is an identifier and parameter sn is a sequence number. The set of outputs is $O = \{\mathsf{OK}, \mathsf{NOK}\}$. The set of states is given by $Q = \mathbb{N} \times \mathbb{N} \times \mathbb{B}$, where the two natural numbers record the current values of id and sn, respectively, and the boolean value denotes whether the component has been initialized. The initial state is $q_0 = (0, 0, \mathsf{F})$. The transition relation contains the following transitions, for all $id, sn, id', sn' \in \mathbb{N}$:

$$
\begin{array}{lll}
(id, sn, \mathsf{F}) & \xrightarrow{REQ(id',sn')/\mathsf{OK}} & (id', sn', \mathsf{T}) \\
(id, sn, \mathsf{T}) & \xrightarrow{REQ(id',sn')/\mathsf{OK}} & (id', sn', \mathsf{T}) \text{ if } id' = id \text{ and } sn' = sn + 1 \\
(id, sn, \mathsf{T}) & \xrightarrow{REQ(id',sn')/\mathsf{NOK}} & (id, sn, \mathsf{T}) \text{ otherwise}
\end{array}
$$

With the first transition the "current" session is initialized by storing the id and sn received in the request message. If in any subsequent request the id of the "current" session is used in combination with the successor of the sequence number sn, an OK output is produced, otherwise a NOK output is returned.

Since infinitely many combinations of concrete values need to be handled, e.g. $REQ(0, 0)$, $REQ(1, 0)$, and $REQ(1, 1)$, application of the L^* algorithm is impossible. To infer the machine, we place a mapper module in between the *learner* and the implementation that abstracts the set of concrete parameter values to (small) finite sets of abstract values, see Figure 3.1.

Concrete symbols of form $REQ(id, sn)$ are abstracted to symbols of form $REQ(\mathrm{ID}, \mathrm{SN})$, where ID and SN are from a small domain, say $\{\mathrm{CUR}, \mathrm{OTHER}\}$. We abstract the parameter value id by CUR if id is the identifier of the "current" session, and by OTHER otherwise. We abstract the parameter sn in a similar way: to CUR if it is the successor of the current sequence number, and

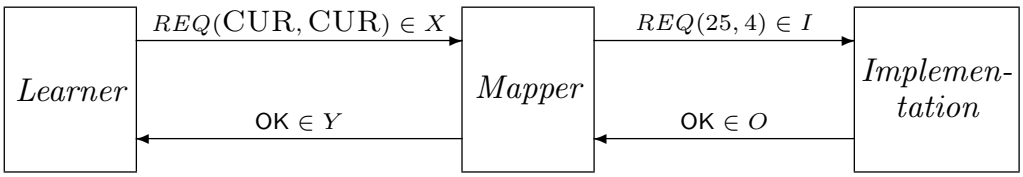


Figure 3.1: *Introduction of mapper component*

to OTHER otherwise. Thus, for instance, input string $REQ(25, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\text{CUR}, \text{CUR})$ $REQ(\text{CUR}, \text{OTHER})$, whereas the input string $REQ(25, 4)$ $REQ(42, 5)$ is abstracted to $REQ(\text{CUR}, \text{CUR})$ $REQ(\text{OTHER}, \text{CUR})$. The resulting abstraction is not "state-free", as it depends on the current values of the session. The mapper records these values in its state. □

In general, in order to learn a "large" Mealy machine $\mathcal{M}$, we place a mapper in between the implementation and the *learner*, which translates the concrete inputs in I to the abstract inputs in X, the concrete outputs in O to the abstract outputs in Y, and vice versa. This will allow us to reduce the task of the *learner* to inferring a "small" Mealy machine with alphabet X and Y, which sometimes is an over-approximation of $\mathcal{M}$, see Section 3.1.4. The next subsection formalizes the concept of a mapper and establishes some technical lemmas. After that, in Subsection 3.1.5, we show how we can turn the abstract model that the *learner* learns in the setup of Figure 3.1, into a correct model for the Mealy machine of the implementation.

3.1.1 Mappers

The behavior of the intermediate component is fully determined by the notion of a *mapper*. A mapper encompasses both concrete and abstract sets of input and output symbols, a set of states, an initial state, a transition function that tells us how the occurrence of a concrete symbol affects the state, and an abstraction function which, depending on the state, maps concrete to abstract symbols.

Definition 3.1 (Mapper) A *mapper* for a set of inputs I and a set of outputs O is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \mathit{abstr} \rangle$, where

- I and O are disjoint sets of *concrete input and output symbols*,
- X and Y are finite sets of *abstract input and output symbols*, and
- $\mathit{abstr} : R \times (I \cup O) \to (X \cup Y)$, referred to as the *abstraction function*, respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \mathit{abstr}(r, a) \in X$.

By definition of Mealy machines, R is the set of states and δ is the update function.

So a mapper is a Mealy machine $\mathcal{A}$ in which the concrete inputs I and outputs O of the implementation act as inputs, and the abstract inputs X and outputs Y used by the *learner* act as outputs. Since for each concrete symbol and each state of the mapper there is a unique abstract symbol, Mealy machine $\mathcal{A}$ is deterministic. An alternative definition, that would be closer to the intuitions reflected in Figure 3.1,

would take X and O as the inputs of the mapper, and I and Y as the outputs. However, such a Mealy machine would in general not be deterministic, and this would complicate subsequent definitions and proofs. Technically, a mapper is just a transducer in the sense of [116], which transforms concrete actions into abstract actions. In fact, the natural notion of composition of mappers is just the standard definition of composition for transducers [116].

Example 3.2 (A mapper for $\mathcal{M}_{COM}$) We define $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, abstr \rangle$, a mapper for the Mealy machine $\mathcal{M}_{COM}$ of Example 3.1. The sets I and O of the mapper are the same as for $\mathcal{M}_{COM}$. The set of abstract input symbols is

$$\begin{aligned} X \quad = \quad & \{REQ(\text{CUR}, \text{CUR}), REQ(\text{CUR}, \text{OTHER}), REQ(\text{OTHER}, \text{CUR}), \\ & REQ(\text{OTHER}, \text{OTHER})\}, \end{aligned}$$

and the set of abstract output symbols Y equals the set of concrete outputs O. The mapper records the current values of id and sn in its state: $R = \{\bot\} \cup (\mathbb{N} \times \mathbb{N})$. Initially, no values for id and sn have been selected: $r_0 = \bot$. The state of the mapper only changes when a $REQ(id, sn)$ input arrives in the initial state, or when id is the current session identifier and sn the successor of the current sequence number:

$$\begin{aligned} \delta(\bot, REQ(id, sn)) &= (id, sn) \\ \delta((id, sn), REQ(id', sn')) &= (id', sn') \text{ if } id' = id \wedge sn' = sn + 1 \\ \delta((id, sn), REQ(id', sn')) &= (id, sn) \text{ if } id' \neq id \vee sn' \neq sn + 1 \end{aligned}$$

Output actions do not change the state of the mapper: $\delta(r, o) = r$, for $r \in R$ and $o \in O$. In the initial state the abstraction function maps all parameter values to CUR:

$$abstr(\bot, REQ(id, sn)) \quad = \quad REQ(\text{CUR}, \text{CUR}).$$

The abstraction function forgets the concrete parameter values of any subsequent request and only records whether they are correct or not:

$$abstr((id, sn), REQ(id', sn')) \quad = \quad REQ(\text{ID}, \text{SN}),$$

where

$$\begin{aligned} \text{ID} \quad &= \quad \textbf{if } id' = id \textbf{ then } \text{CUR} \textbf{ else } \text{OTHER}, \text{ and} \\ \text{SN} \quad &= \quad \textbf{if } sn' = sn + 1 \textbf{ then } \text{CUR} \textbf{ else } \text{OTHER}. \end{aligned}$$

For outputs *abstr* acts as the identity function. □

3.1.2 The Abstraction Operator

A mapper allows us to abstract a Mealy machine with concrete symbols in I and O into a Mealy machine with abstract symbols in X and Y, and conversely, via an adjoint operator, to concretize a Mealy machine with symbols in X and Y into a Mealy machine with symbols in I and O. First we show how an abstract Mealy machine can be built from a mapper and a concrete Mealy machine, and explore

some properties of this construction. Basically, the *abstraction* of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert concrete symbols into abstract ones.

Definition 3.2 (Abstraction) Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \mathit{abstr} \rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the *abstraction of* $\mathcal{M}$ *via* $\mathcal{A}$, is the Mealy machine $\langle X, Y \cup \{\bot\}, Q \times R, (q_0, r_0), \rightarrow \rangle$, where $\bot$ is a fresh abstract output symbol and $\rightarrow$ is given by the rules

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i/x} r' \xrightarrow{o/y} r''}{(q,r) \xrightarrow{x/y} (q',r'')} \qquad \frac{\nexists i \in I : r \xrightarrow{i/x}}{(q,r) \xrightarrow{x/\bot} (q,r)}$$

The first rule says that a state (q, r) of the abstraction has an outgoing x-transition for each transition $q \xrightarrow{i/o} q'$ of $\mathcal{M}$ with $\mathit{abstr}(r, i) = x$. In this case, there exist unique r', r'' and y such that $r \xrightarrow{i/x} r' \xrightarrow{o/y} r''$ in the mapper. An x-transition in state (q, r) then leads to state (q', r'') and produces output y. The second rule in the definition is required to ensure that the abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ is input enabled. Given a state (q, r) of the mapper, it may occur that for some abstract input symbol x there exists no corresponding concrete input symbol i with $\mathit{abstr}(r, i) = x$. In this case, an input x triggers the special "undefined" output symbol $\bot$ and leaves the state unchanged.

Example 3.3 (Abstraction of $\mathcal{M}_{COM}$) The abstraction $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ of our example Mealy machine $\mathcal{M}_{COM}$ has the same abstract input and output symbols as mapper $\mathcal{A}$, except for an additional "undefined" abstract output symbol $\bot$. States of the abstract Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ are pairs (q, r) where q is a state of Mealy machine $\mathcal{M}_{COM}$ and r is a state of mapper $\mathcal{A}$. The initial state is $((0, 0, \mathsf{F}), \bot)$. We have the following transitions, for all $sn, id \in \mathbb{N}$ (only transitions reachable from the initial state are listed):

$$\begin{aligned}
((0,0,\mathsf{F}),\bot) &\xrightarrow{\mathit{REQ}(\mathrm{CUR},\mathrm{CUR})/\mathsf{OK}} ((id,sn,\mathsf{T}),(id,sn)) \\
((0,0,\mathsf{F}),\bot) &\xrightarrow{\mathit{REQ}(\mathrm{CUR},\mathrm{OTHER})/\bot} ((0,0,\mathsf{F}),\bot) \\
((0,0,\mathsf{F}),\bot) &\xrightarrow{\mathit{REQ}(\mathrm{OTHER},\mathrm{CUR})/\bot} ((0,0,\mathsf{F}),\bot) \\
((0,0,\mathsf{F}),\bot) &\xrightarrow{\mathit{REQ}(\mathrm{OTHER},\mathrm{OTHER})/\bot} ((0,0,\mathsf{F}),\bot) \\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{\mathit{REQ}(\mathrm{CUR},\mathrm{CUR})/\mathsf{OK}} ((id,sn+1,\mathsf{T}),(id,sn+1)) \\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{\mathit{REQ}(\mathrm{CUR},\mathrm{OTHER})/\mathsf{NOK}} ((id,sn,\mathsf{T}),(id,sn)) \\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{\mathit{REQ}(\mathrm{OTHER},\mathrm{CUR})/\mathsf{NOK}} ((id,sn,\mathsf{T}),(id,sn)) \\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{\mathit{REQ}(\mathrm{OTHER},\mathrm{OTHER})/\mathsf{NOK}} ((id,sn,\mathsf{T}),(id,sn))
\end{aligned}$$

Observe that, by the second rule in Definition 3.2, the abstract inputs $\mathit{REQ}(\mathrm{CUR}, \mathrm{OTHER})$, $\mathit{REQ}(\mathrm{OTHER}, \mathrm{CUR})$, and $\mathit{REQ}(\mathrm{OTHER}, \mathrm{OTHER})$ in the initial state trigger an output $\bot$, since in this state all concrete input actions are mapped to

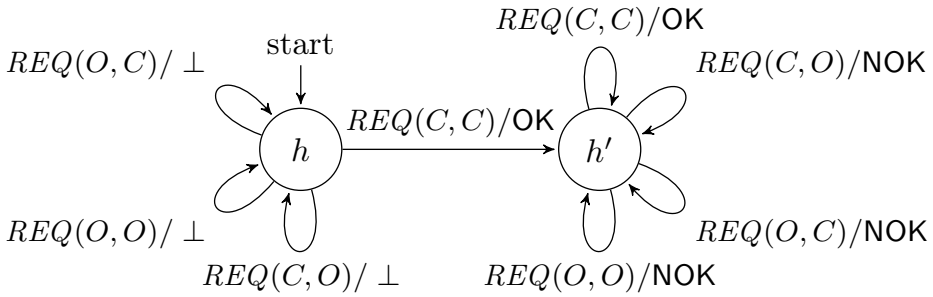


Figure 3.2: *Minimal Mealy machine $\mathcal{H}_{COM}$ equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$*

$REQ(\text{CUR}, \text{CUR})$. Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is infinite state, but it is not hard to see that it is observation equivalent to the finite, deterministic Mealy machine $\mathcal{H}_{COM}$ displayed in Figure 3.2. □

The abstraction function of a mapper can be lifted to observations in a straightforward manner: every concrete input or output string can be turned into an abstract string by stepwise transforming every symbol according to *abstr*. First, we need some notation. Given two sequences u and s of equal length, $zip(u, s)$ is the sequence obtained by zipping them together. Function *zip* is inductively defined as follows:

$$\begin{aligned} zip(\epsilon, \epsilon) &= \epsilon \\ zip(i\ u, o\ s) &= i\ o\ zip(u, s) \end{aligned}$$

Conversely, given any sequence w with an even number of elements, $odd(w)$ and $even(w)$ are the subsequences obtained by picking all the odd resp. even elements from w:

$$\begin{aligned} odd(\epsilon) &= \epsilon \\ odd(a\ b\ w) &= a\ odd(w) \\ even(\epsilon) &= \epsilon \\ even(a\ b\ w) &= b\ even(w) \end{aligned}$$

By induction it follows that $zip(odd(w), even(w)) = w$.

Definition 3.3 (Abstraction of observations) Let $\mathcal{A}$ be a mapper. Then function $\tau_{\mathcal{A}}$, which maps concrete observations over I and O to abstract observations over X and Y, is defined by

$$\tau_{\mathcal{A}}(u, s) : I^* \times O^* \to X^* \times Y^* = (odd(w), even(w)), \text{ where } w = abstr(r_0, zip(u, s)).$$

For a given mapper $\mathcal{A}$, the abstraction operator on Mealy machines is of course closely related to the abstraction operator on observations. The connection is formally established in Claim 1 below. Using the claim, we link observations of $\mathcal{M}$ to observations of $\alpha_{\mathcal{A}}(\mathcal{M})$ in Lemma 3.1.

Claim 1. *Suppose $q \overset{u/s}{\Rightarrow} q'$ is a transition of Mealy machine $\mathcal{M}$, $r' = \delta(r, zip(u, s))$, $w = abstr(r, zip(u, s))$, $u' = odd(w)$ and $s' = even(w)$. Then $(q, r) \overset{u'/s'}{\Rightarrow} (q', r')$ is a transition of $\alpha_{\mathcal{A}}(\mathcal{M})$.*

Proof. By induction on the length of u.
Basis: $|u| = 0$. Then $u = \epsilon$ and because $q \stackrel{u/s}{\Rightarrow} q'$ implies $|u| = |s|$, also $s = \epsilon$. Since $q \stackrel{\epsilon/\epsilon}{\Rightarrow} q'$, it follows that $q = q'$. Furthermore, $r' = \delta(r, zip(\epsilon, \epsilon)) = \delta(r, \epsilon) = r$, $w = abstr(r, zip(\epsilon, \epsilon)) = abstr(r, \epsilon) = \epsilon$, $u' = odd(\epsilon) = \epsilon$ and $s' = even(\epsilon) = \epsilon$. This implies $(q, r) \stackrel{u'/s'}{\Rightarrow} (q', r')$, as required.
Induction step: Assume $u = i\ \bar{u}$, where $i \in I$ and $\bar{u}$ is of length n. Then we can write $s = o\ \bar{s}$, where $o \in O$ and $\bar{s}$ is of length n. Since $q \stackrel{u/s}{\Rightarrow} q'$, there exists a state q'' such that

$$q \xrightarrow{i/o} q'' \quad \wedge \quad q'' \stackrel{\bar{u}/\bar{s}}{\Rightarrow} q'.$$

Let $r_1 = \delta(r, i)$ and $r_2 = \delta(r_1, o)$. We infer

$$r' \quad = \quad \delta(r, zip(u, s)) = \delta(r, i\ o\ zip(\bar{u}, \bar{s})) = \delta(r_1, o\ zip(\bar{u}, \bar{s})) = \delta(r_2, zip(\bar{u}, \bar{s})).$$

Let $w' = abstr(r_2, zip(\bar{u}, \bar{s}))$, $u'' = odd(w')$ and $s'' = even(w')$. By induction hypothesis,

$$(q'', r_2) \stackrel{u''/s''}{\Rightarrow} (q', r') \tag{1}$$

is a transition of $\alpha_{\mathcal{A}}(\mathcal{M})$. Let $x = abstr(r, i)$ and $y = abstr(r_1, o)$. Then by the first rule in the definition of $\alpha_{\mathcal{A}}(\mathcal{M})$,

$$(q, r) \xrightarrow{x/y} (q'', r_2). \tag{2}$$

We infer

$$\begin{aligned}
w &= abstr(r, zip(u, s)) = abstr(r, i\ o\ zip(\bar{u}, \bar{s})) = x\ abstr(r_1, o\ zip(\bar{u}, \bar{s})) \\
&= x\ y\ abstr(r_2, zip(\bar{u}, \bar{s})) = x\ y\ w', \\
u' &= odd(w) = odd(x\ y\ w') = x\ odd(w') = x\ u'', && (3) \\
s' &= even(w) = even(x\ y\ w') = y\ even(w') = y\ s''. && (4)
\end{aligned}$$

Combination (1), (2), (3) and (4) now gives $(q, r) \stackrel{u'/s'}{\Rightarrow} (q', r')$, as required. □

Lemma 3.1 Suppose $(u, s) \in obs_{\mathcal{M}}$. Then $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$.

Proof. Let $r' = \delta(r_0, zip(u, s))$, $w = abstr(r_0, zip(u, s))$, $u' = odd(w)$ and $s' = even(w)$. Then

$$\begin{aligned}
(u, s) \in obs_{\mathcal{M}} \quad &\Rightarrow \quad \text{(Definition of } obs\text{)} \\
\exists q' : q_0 \stackrel{u/s}{\Rightarrow} q' \quad &\Rightarrow \quad \text{(Claim 1)} \\
\exists q' : (q_0, r_0) \stackrel{u'/s'}{\Rightarrow} (q', r') \quad &\Rightarrow \quad \text{(Definition of } obs\text{)} \\
(u', s') \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} \quad &\Rightarrow \quad \text{(Definition } \tau_{\mathcal{A}}\text{)} \\
\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} &
\end{aligned}$$

□

Lemma 3.1 allows us to construct, for each concrete observation of $\mathcal{M}$, a unique abstract observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ that corresponds to it. Given an abstract observation, there may in general be many corresponding concrete observations. However, for abstract observations containing the undefined output symbol $\perp$, there exists no corresponding concrete observation since by definition $\tau_{\mathcal{A}}(u, s)$ contains no $\perp$'s. According to the following claim and lemma, for each observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ without output $\perp$ there exists at least one corresponding observation of $\mathcal{M}$.

Claim 2. *Suppose* (u, s) *is an observation over* X *and* Y*, and* $(q, r) \stackrel{u/s}{\Rightarrow} (q', r')$ *is a transition of* $\alpha_{\mathcal{A}}(\mathcal{M})$*. Then there exists an observation* (u', s') *such that* $q \stackrel{u'/s'}{\Rightarrow} q'$*,* $u = odd(w)$ *and* $s = even(w)$*, where* $w = abstr(r, zip(u', s'))$*.*

Proof. By a routine induction on the length of u.
Basis: $|u| = 0$. Then $u = \epsilon$ and because (u, s) is an observation over X and Y, also $s = \epsilon$. But this implies that $q' = q$. Let $u' = s' = \epsilon$. Then $q \stackrel{u'/s'}{\Rightarrow} q'$, $w = \epsilon$, and thus $u = odd(w)$ and $s = even(w)$, as required.
Induction step: Assume $u = x\bar{u}$ where $x \in X$. Since (u, s) is an observation over X and Y, we can write $s = y\bar{s}$ where $y \in Y$. Then $(\bar{u}, \bar{s})$ is also an observation over X and Y. By definition of $\stackrel{u'/s'}{\Rightarrow}$, there exists an intermediate state (q'', r'') such that $(q, r) \xrightarrow{x/y} (q'', r'') \stackrel{\bar{u}/\bar{s}}{\Rightarrow} (q', r')$. By the first rule in the definition of $\alpha_{\mathcal{A}}(\mathcal{M})$, there exist concrete actions i and o, and a state $\bar{r}$ such that $q \xrightarrow{i/o} q''$ and $r \xrightarrow{i/x} \bar{r} \xrightarrow{o/y} r''$. Moreover, by induction hypothesis, there exists an observation $(\bar{u}', \bar{s}')$ such that $q'' \stackrel{\bar{u}'/\bar{s}'}{\Rightarrow} q'$, $\bar{u} = odd(\bar{w})$ and $\bar{s} = even(\bar{w})$, where $\bar{w} = abstr(r'', zip(\bar{u}', \bar{s}'))$. Let $u' = i\bar{u}'$, $s' = o\bar{s}'$, and $w = xy\bar{w}$. Then $q \stackrel{u'/s'}{\Rightarrow} q'$. Moreover:

$$\begin{aligned} odd(w) &= odd(xy\bar{w}) = x\ odd(\bar{w}) = x\bar{u} = u, \\ even(w) &= even(xy\bar{w}) = y\ even(\bar{w}) = y\bar{s} = s, \text{ and} \\ w &= xy\bar{w} = xyabstr(r'', zip(\bar{u}', \bar{s}')) = abstr(r, i\ o\ zip(\bar{u}', \bar{s}')) \\ &= abstr(r, zip(i\bar{u}', o\bar{s}')) = abstr(r, zip(u', s')). \end{aligned}$$

□

Lemma 3.2 Suppose $(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$ is an observation over X and Y. Then $\exists (u', s') \in obs_{\mathcal{M}} : \tau_{\mathcal{A}}(u', s') = (u, s)$.

Proof. By assumption, $\alpha_{\mathcal{A}}(\mathcal{M})$ has a state (q, r) such that $(q_0, r_0) \stackrel{u/s}{\Rightarrow} (q, r)$. By Claim 2, there is an observation (u', s') such that $q_0 \stackrel{u'/s'}{\Rightarrow} q$, $u = odd(w)$ and $s = even(w)$, where $w = abstr(r_0, zip(u', s'))$. Thus $\tau_{\mathcal{A}}(u', s') = (u, s)$ and $(u', s') \in obs_{\mathcal{M}}$, as required. □

3.1.3 The Concretization Operator

We now define the *concretization* operator, which is the adjoint of the abstraction operator. For a given mapper $\mathcal{A}$, the corresponding concretization operator turns any abstract Mealy machine with symbols in X and Y into a concrete Mealy

machine with symbols in I and O. Basically, the concretization of Mealy machine $\mathcal{H}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert abstract symbols into concrete ones.

Definition 3.4 (Concretization) Let $\mathcal{H} = \langle X, Y \cup \{\bot\}, H, h_0, \rightarrow\rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, abstr\rangle$ be a mapper for I and O. Then $\gamma_{\mathcal{A}}(\mathcal{H})$, the *concretization* of $\mathcal{H}$ via $\mathcal{A}$, is the Mealy machine $\langle I, O \cup \{\bot\}, R \times H, (r_0, h_0), \rightarrow\rangle$, where $\rightarrow$ is given by the rules

$$\frac{r \xrightarrow{i/x} r' \xrightarrow{o/y} r'',\ h \xrightarrow{x/y} h'}{(r,h) \xrightarrow{i/o} (r'',h')} \qquad \frac{r \xrightarrow{i/x} r',\ h \xrightarrow{x/y} h',\ \nexists o \in O : r' \xrightarrow{o/y}}{(r,h) \xrightarrow{i/\bot} (r,h)}$$

States of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ are pairs (r,h) of a state h of the hypothesis and a state r of the mapper. Each transition $h \xrightarrow{x/y} h'$ of the hypothesis corresponds to potentially many transitions of the concretization: (r,h) has an outgoing i/o transition whenever $abstr(r,i) = x$ and $abstr(r',o) = y$, where r' is the unique state such that $r \xrightarrow{i} r'$. The second rule in the definition is required to ensure the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is input enabled. Consider a state (r,h) of the concretization and a concrete input i. Since $\mathcal{A}$ is deterministic and input enabled, there exists a unique state r' such that $r \xrightarrow{i} r'$. Let $x = abstr(r,i)$ be the corresponding abstract input. Since $\mathcal{H}$ is input enabled, there also exists a state h' and an abstract output y such that $h \xrightarrow{x/y} h'$. However, there does not necessarily exist an output o with $abstr(r',o) = y$. This means that the first rule cannot always be applied to infer an outgoing i-transition of state (r,h). In order to ensure input enabledness, the second rule is used in this case to introduce a transition with "undefined" output $\bot$ that leaves the state (r,h) unchanged.

Example 3.4 (Concretization of $\mathcal{H}_{COM}$) Let us now concretize the abstract Mealy machine $\mathcal{H}_{COM}$ of Figure 3.2, which is observation equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$. The Mealy machine $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ has the same concrete input and output symbols as $\mathcal{M}_{COM}$, except for the additional output $\bot$. States of the concretization are pairs of states of $\mathcal{A}$ and states of $\mathcal{H}_{COM}$. The initial state is $(\bot,h)$. We have the following transitions, for all $id, id', sn, sn' \in \mathbb{N}$ with $id' \neq id$ and $sn' \neq sn+1$ (only transitions reachable from the initial state are listed):

$$\begin{aligned}
(\bot,h) &\xrightarrow{REQ(id,sn)/\mathsf{OK}} ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id,sn+1)/\mathsf{OK}} ((id,sn+1),h') \\
((id,sn),h') &\xrightarrow{REQ(id,sn')/\mathsf{NOK}} ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id',sn+1)/\mathsf{NOK}} ((id,sn),h') \\
((id,sn),h') &\xrightarrow{REQ(id',sn')/\mathsf{NOK}} ((id,sn),h')
\end{aligned}$$

Note that the transitions with output $\bot$ in $\mathcal{H}_{COM}$ play no role in $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ since there exists no concrete output of $\mathcal{A}$ that is abstracted to $\bot$: the only use of these transitions is to make $\mathcal{H}_{COM}$ input enabled. Also note that in this specific

example the second rule of Definition 3.4 does not play a role, since *abstr* acts as the identity function on outputs. The reader may check that $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ is observation equivalent to $\mathcal{M}_{COM}$. □

Claim 3 and Lemma 3.3 below link the behavior of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to the behavior of $\mathcal{H}$.

Claim 3. *Let (u, s) be an observation over inputs I and outputs O, let $r \in R$, $r' = \delta(r, zip(u, s))$, $w = abstr(r, zip(u, s))$, $u' = odd(w)$ and $s' = even(w)$. Then $h \overset{u'/s'}{\Rightarrow} h'$ is a transition of $\mathcal{H}$ iff $(r, h) \overset{u/s}{\Rightarrow} (r', h')$ is a transition of $\gamma_{\mathcal{A}}(\mathcal{H})$.*

Proof. Proof by induction on length of u.
Basis: $|u| = 0$. Then $u = \epsilon$ and, because (u, s) is an observation, also $s = \epsilon$. Hence $r' = \delta(r, zip(\epsilon, \epsilon)) = \delta(r, \epsilon) = r$, $w = abstr(r, zip(\epsilon, \epsilon)) = abstr(r, \epsilon) = \epsilon$, $u' = odd(\epsilon) = \epsilon$ and $s' = even(\epsilon) = \epsilon$. We infer

$$h \overset{u'/s'}{\Rightarrow} h' \text{ iff } h \overset{\epsilon/\epsilon}{\Rightarrow} h' \text{ iff } h = h' \text{ iff } (r, h) \overset{\epsilon/\epsilon}{\Rightarrow} (r, h') \text{ iff } (r, h) \overset{u/s}{\Rightarrow} (r', h').$$

Induction step: Assume $u = i\ \bar{u}$, where $\bar{u}$ is of length n. Then we can write $s = o\ \bar{s}$, where $\bar{s}$ is of length n. Let $r_1 = \delta(r, i)$ and $r_2 = \delta(r_1, o)$. Then

$$r' \quad = \quad \delta(r, zip(u, s)) = \delta(r, i\ o\ zip(\bar{u}, \bar{s})) = \delta(r_1, o\ zip(\bar{u}, \bar{s})) = \delta(r_2, zip(\bar{u}, \bar{s})).$$

Let $w' = abstr(r_2, zip(\bar{u}, \bar{s}))$, $u'' = odd(w')$, $s'' = even(w')$, $x = abstr(r, i)$ and $y = abstr(r_1, o)$. We infer

$$\begin{aligned}
w &= abstr(r, zip(u, s)) = abstr(r, i\ o\ zip(\bar{u}, \bar{s})) = x\ abstr(r_1, o\ zip(\bar{u}, \bar{s})) \\
&= x\ y\ abstr(r_2, zip(\bar{u}, \bar{s})) = x\ y\ w', \\
u' &= odd(w) = odd(x\ y\ w') = x\ odd(w') = x\ u'', \\
s' &= even(w) = even(x\ y\ w') = y\ even(w') = y\ s''.
\end{aligned}$$

Thus

$$\begin{aligned}
h \overset{u'/s'}{\Rightarrow} h' &\quad \Leftrightarrow \\
\exists h'' : h \xrightarrow{x/y} h'' \overset{u''/s''}{\Rightarrow} h' &\quad \Leftrightarrow \quad \text{(first rule in definition } \gamma_{\mathcal{A}}(\mathcal{H}) \text{ and IH)} \\
\exists h'' : (r, h) \xrightarrow{i/o} (r_2, h'') \overset{\bar{u}/\bar{s}}{\Rightarrow} (r', h') &\quad \Leftrightarrow \\
(r, h) \overset{u/s}{\Rightarrow} (r', h') &
\end{aligned}$$

□

Lemma 3.3 Let (u, s) be an observation over inputs I and outputs O. Then $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$ iff $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$.

Proof. Let $w = abstr(r_0, zip(u, s))$, $u' = odd(w)$, $s' = even(w)$, and $r' = \delta(r_0, zip(u, s))$. We infer

$$\begin{aligned}
\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}} &\quad \Leftrightarrow \\
(u', s') \in obs_{\mathcal{H}} &\quad \Leftrightarrow
\end{aligned}$$

$$\begin{aligned} \exists h' : h_0 \overset{u'/s'}{\Rightarrow} h' \quad &\Leftrightarrow \quad \text{(by Claim 3)} \\ \exists h' : (r_0, h_0) \overset{u/s}{\Rightarrow} (r', h') \quad &\Leftrightarrow \\ (u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})} & \end{aligned}$$

□

The following theorem, which builds on the previous lemmas in this section, establishes the duality of the concretization and abstraction operators.

Theorem 3.1 $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.

Proof. Suppose $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. Let $(u, s) \in obs_{\mathcal{M}}$. It suffices to prove $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. By Lemma 3.1, $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. By the assumption, $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$. Hence, by Lemma 3.3, $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. □

Example 3.5 The example of Figure 3.3 shows that the converse of Theorem 3.1 does not hold. All the Mealy machines in the example have just a single state.

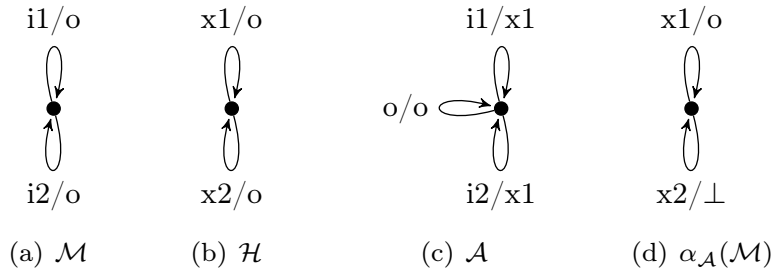


Figure 3.3: *Counterexample for $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ implies $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$*

Mapper $\mathcal{A}$ abstracts both concrete inputs $i1$ and $i2$ to the abstract input $x1$. However, there is also another abstract input $x2$, which messes things up. The reader may check that $\gamma_{\mathcal{A}}(\mathcal{H}) \approx \mathcal{M}$. However, it is not the case that $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, since $\alpha_{\mathcal{A}}(\mathcal{M})$ may generate an output $\perp$ whereas $\mathcal{H}$ only generates output o. □

It turns out that the converse of Theorem 3.1 holds if we add the assumption that $\alpha_{\mathcal{A}}(\mathcal{M})$ does not generate the undefined output $\perp$.

Theorem 3.2 Suppose $\alpha_{\mathcal{A}}(\mathcal{M})$ has no observations with output $\perp$. Then $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ implies $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$.

Proof. Suppose $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Let $(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. It suffices to prove $(u, s) \in obs_{\mathcal{H}}$. By assumption, (u, s) is an observation over X and Y. Hence, by Lemma 3.2, there exists $(u', s') \in obs_{\mathcal{M}}$ with $\tau_{\mathcal{A}}(u', s') = (u, s)$. By the assumption, $(u', s') \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. Hence, by Lemma 3.3, $\tau_{\mathcal{A}}(u', s') = (u, s) \in obs_{\mathcal{H}}$. □

In fact, the above result can be slightly strengthened: in general, if $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ and one takes any observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ and removes the undefined inputs, then one obtains an observation of $\mathcal{H}$.

3.1.4 Learned Models as Over-Approximations

For many applications it is difficult to predict exactly which output will occur when. In these cases it makes sense to define mappers that abstract away information from the implementation. With such mappers we will not learn a Mealy machine that is observation equivalent to the Mealy machine of the implementation, but rather a nondeterministic over-approximation of it. In order to allow for such over-approximations, we have replaced Angluin's *equivalence queries* by *inclusion queries* in our learning framework.

Example 3.6 In order to illustrate this, we consider an alternative mapper for $\mathcal{M}_{COM}$:

$$\mathcal{A}' \quad = \quad \langle I \cup O, X' \cup O, \{\bot\} \cup \mathbb{N}, \bot, \delta', \mathit{abstr}' \rangle.$$

The sets I and O are the same as for $\mathcal{M}_{COM}$. Mapper $\mathcal{A}'$ only records the selected value of the identifier and ignores the sequence number parameter. The state of $\mathcal{A}'$ only changes when the first $REQ(id, sn)$ input arrives:

$$\begin{aligned} \delta'(\bot, REQ(id, sn)) &= id, \\ \delta'(id, REQ(id', sn')) &= id. \end{aligned}$$

Output actions do not change the state of $\mathcal{A}'$: $\delta'(r, o) = r$, for $r \in \{\bot\} \cup \mathbb{N}$ and $o \in O$. There are two abstract input symbols: $X' = \{REQ(\mathrm{CUR}), REQ(\mathrm{OTHER})\}$. In the initial state the abstraction function maps to $REQ(\mathrm{CUR})$, and for subsequent actions it only records whether the identifier is correct:

$$\begin{aligned} \mathit{abstr}'(\bot, REQ(id, sn)) &= REQ(\mathrm{CUR}), \\ \mathit{abstr}'(id, REQ(id', sn')) &= REQ(\mathrm{ID}), \end{aligned}$$

where ID = **if** $id' = id$ **then** CUR **else** OTHER. For outputs abstr' acts as the identity function.

States of the abstract Mealy machine $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$ are pairs (q, r) where q is a state of Mealy machine $\mathcal{M}_{COM}$ and r is a state of mapper $\mathcal{A}'$. The initial state is $((0, 0, \mathsf{F}), \bot)$. We have the following transitions, for all $sn, id \in \mathbb{N}$ (only transitions reachable from the initial state are listed):

$$\begin{aligned} ((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{CUR})/\mathsf{OK}} ((id,sn,\mathsf{T}),id) \\ ((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{OTHER})/\bot} ((0,0,\mathsf{F}),\bot) \\ ((id,sn,\mathsf{T}),id) &\xrightarrow{REQ(\mathrm{CUR})/\mathsf{OK}} ((id,sn+1,\mathsf{T}),id) \\ ((id,sn,\mathsf{T}),id) &\xrightarrow{REQ(\mathrm{CUR})/\mathsf{NOK}} ((id,sn,\mathsf{T}),id) \\ ((id,sn,\mathsf{T}),id) &\xrightarrow{REQ(\mathrm{OTHER})/\mathsf{NOK}} ((id,sn,\mathsf{T}),id) \end{aligned}$$

The last three transitions correspond to the cases in which, respectively, both the identifier and sequence number of a request are correct, the identifier is correct but the sequence number is not, and the identifier is incorrect. It is easy to see that $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$ is behaviorally equivalent to the nondeterministic Mealy machine

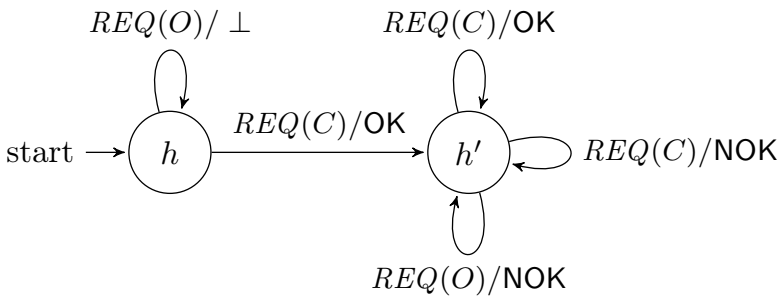


Figure 3.4: *Minimal Mealy machine $\mathcal{H}'_{COM}$ equivalent to $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$*

$\mathcal{H}'_{COM}$ displayed in Figure 3.4: all states in the set $\{((id, sn, \mathsf{T}), id) \mid sn, id \in \mathbb{N}\}$ are equivalent (bisimilar).

The concretization $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has the following transitions, for all $id, id', sn \in \mathbb{N}$ with $id' \neq id$ (only transitions reachable from the initial state are listed):

$$
\begin{aligned}
(\bot, h) &\xrightarrow{REQ(id,sn)/\mathsf{OK}} (id, h') \\
(id, h') &\xrightarrow{REQ(id,sn)/\mathsf{OK}} (id, h') \\
(id, h') &\xrightarrow{REQ(id,sn)/\mathsf{NOK}} (id, h') \\
(id, h') &\xrightarrow{REQ(id',sn)/\mathsf{NOK}} (id, h')
\end{aligned}
$$

By Theorem 3.1, we have $\mathcal{M}_{COM} \leq \gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$. This time $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ is an over-approximation of $\mathcal{M}_{COM}$ since, for instance, $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has a trace $REQ(1,2)/\mathsf{OK}\ REQ(1,2)/\mathsf{OK}$, which is not allowed by $\mathcal{M}_{COM}$. □

Whenever we succeed to learn a hypothesis $\mathcal{H}$ such that $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ and $\gamma_{\mathcal{A}}(\mathcal{H})$ is behavior deterministic, then $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$ by Lemma 2.1. This means that, even though we have used a mapper component, we have lost no information. If we use LearnLib to construct hypothesis $\mathcal{H}$, $\mathcal{H}$ will be a deterministic Mealy machine. We will now present some conditions under which $\gamma_{\mathcal{A}}$ preserves determinism.

Let $y \in Y$ be an abstract output. Then mapper $\mathcal{A}$ is *output-predicting* for y if, for all concrete outputs $o, o' \in O$ and for all mapper states $r \in R$, $abstr(r, o) = y$ and $abstr(r, o') = y$ implies $o = o'$. We call $\mathcal{A}$ *output-predicting* if it is output-predicting for all $y \in Y$, that is, *abstr* is injective on outputs for fixed r.

Using the next lemma, which follows immediately from the definitions, we may infer that $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.

Lemma 3.4 Suppose $\mathcal{H}$ is deterministic and $\mathcal{A}$ is output-predicting for all outputs y that occur in transitions of $\mathcal{H}$. Then $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.

Example 3.7 The mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$ introduced in Example 3.2 acts as the identity operation on outputs and is thus trivially output-predicting. This mapper is just right: Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is simple and has only two states, it is deterministic, and if we concretize it again then the resulting Mealy machine $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M}_{COM}))$ is deterministic and observation equivalent to the original $\mathcal{M}_{COM}$. □

In Section 3.4.1, we will see a less trivial application of Lemma 3.4, where it is

used to establish that the concretization of a learned model of the SIP protocol is deterministic.

3.1.5 The Behavior of the Mapper Component

We are now prepared to formalize the ideas of Example 3.1 and establish that, by using an intermediate mapper component, a *learner* can indeed learn a correct model of the behavior of an implementation.

Consider a mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \mathit{abstr} \rangle$. The mapper component that is induced by $\mathcal{A}$ records the current state, which initially is set to r_0. The behavior of the component can informally be described as follows:

- Whenever the component is in a state r and receives an abstract input symbol $x \in X$ from the *learner*, it nondeterministically picks a concrete input symbol $i \in I$ such that $\mathit{abstr}(r, i) = x$, forwards i to the implementation, and jumps to state $\delta(r, i)$. If there exists no concrete input i such that $\mathit{abstr}(r, i) = x$, then the component returns output $\perp$ to the *learner*.
- Whenever the component is in a state r and receives a concrete answer o from the implementation, it forwards the abstract version $\mathit{abstr}(r, o)$ to the *learner* and jumps to state $\delta(r, o)$.
- Whenever the component receives a reset query from the *learner*, it changes its current state to r_0, and forwards a reset query to the implementation.

We claim that, from the perspective of a *learner*, an implementation for $\mathcal{M}$ and a mapper component for $\mathcal{A}$ together behave exactly like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$. Since we have not formalized the notion of behavior for implementation and mapper, the mathematical content of this claim may not be immediately obvious. Clearly, it is routine to describe the behavior of an implementation and a mapper formally as state machines in some concurrency formalism, for instance in Milner's CCS [115] or another process algebra [29]. More precisely, we may define, for each Mealy machine $\mathcal{M}$, a CCS process $\mathsf{Impl}(\mathcal{M}, \mathsf{reset})$, the implementation of $\mathcal{M}$ with resetting action reset, where reset is a fresh name not in $I \cup O$, and for each mapper $\mathcal{A}$ a CCS process $\mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}')$, the implementation of $\mathcal{A}$ with input reset and output $\overline{\mathsf{reset}'}$, where reset and reset' are fresh action names.

The implementation and the mapper may synchronize via actions taken from the set $L = I \cup O \cup \{\mathsf{reset}'\}$. If we compose $\mathsf{Impl}(\mathcal{M}, \mathsf{reset}')$ and $\mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}')$ using the CCS composition operator $|$, and apply the CCS restriction operator $\setminus$ to internalize communications from L between the two processes, the resulting CCS process is observation congruent (weakly bisimilar) to the CCS process $\mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset})$. In order to avoid confusion, we write $\approx_{wb}$ instead of $=$ (as in [115]) to denote observation congruence:

$$(\mathsf{Impl}(\mathcal{M}, \mathsf{reset}') \mid \mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}')) \setminus L \quad \approx_{wb} \quad \mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset}).$$

It is in this precise, formal sense that one should read the following theorem. The reason why we do not refer to the CCS formalization in the statement and proof of this theorem is that we feel that the resulting notational overhead would obscure rather than clarify.

Theorem 3.3 An implementation for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Proof. Initially, the state of the implementation for $\mathcal{M}$ is q_0 and the current state of the mapper is r_0, which is consistent with the initial state (q_0, r_0) of an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Suppose that the current state is (q, r) and an output query $x \in X$ arrives. If there exists a r' such that $r \xrightarrow{i/x} r'$, then the mapper nondeterministically picks one such concrete i, passes it on to the implementation (which accepts concrete input symbols), and jumps to state r'. In response, the implementation picks a transition $q \xrightarrow{i/o} q'$, jumps to state q' and returns the concrete output symbol $o \in O$ to the mapper. Next, the mapper takes the corresponding transition $r' \xrightarrow{o/y} r''$, forwards y to the *learner*, and jumps to state r''. By inspection of the first transition rule for $\alpha_{\mathcal{A}}(\mathcal{M})$, it follows that the implementation for $\mathcal{M}$ and mapper together behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$ in this case. If there exists no r' such that $r \xrightarrow{i/x} r'$, then the mapper returns output $\perp$ to the *learner*. By inspection of the second transition rule for $\alpha_{\mathcal{A}}(\mathcal{M})$, it follows that the implementation of $\mathcal{M}$ and mapper together again behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Now suppose the mapper receives a reset query from the *learner*. Then the mapper moves to its initial state r_0 and forwards the reset query to the implementation, who also returns to its initial state q_0. This behavior is consistent with the behavior of an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$, which returns to its initial state (q_0, r_0) upon receiving a reset query. □

3.1.6 Mappers and Oracles

In order to learn a model, a *learner* does not only need an implementation allowing it to construct hypotheses, but also an oracle to establish the correctness of these hypotheses. In the previous subsection, we discussed how an implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$ can be constructed out of an implementation of $\mathcal{M}$ and a mapper component for $\mathcal{A}$. We will now discuss how an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ can be obtained.

A first approach, also explored in [4], is to construct an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ from an oracle for $\mathcal{M}$. When the *learner* produces a hypothesis $\mathcal{H}$ for $\alpha_{\mathcal{A}}(\mathcal{M})$, the idea is to forward the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to the oracle for $\mathcal{M}$. There are two cases.

If the concrete hypothesis is incorrect, that is, $\mathcal{M} \not\leq \gamma_{\mathcal{A}}(\mathcal{H})$, then the oracle for $\mathcal{M}$ will produce a counterexample $(u, s) \in obs_{\mathcal{M}} - obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. By Theorem 3.1, we know that $\alpha_{\mathcal{A}}(\mathcal{M}) \not\leq \mathcal{H}$, that is, the abstract hypothesis $\mathcal{H}$ is incorrect. Since $(u, s) \in obs_{\mathcal{M}}$, Lemma 3.1 gives $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. We also know that (u, s) is an observation over I and O. Hence, since $(u, s) \notin obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$, Lemma 3.3 gives $\tau_{\mathcal{A}}(u, s) \notin obs_{\mathcal{H}}$. Thus $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} - obs_{\mathcal{H}}$ is a counterexample that demonstrates that $\mathcal{H}$ is incorrect. We forward this counterexample to the *learner*, in accordance with the required behavior for an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$.

If the concrete hypothesis is correct, that is, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$, then the oracle for $\mathcal{M}$ will produce output *yes*. Due to the example of Figure 3.3, we may not conclude $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, and thus we may not forward the *yes* to the *learner* (except if somehow we manage to infer that $\alpha_{\mathcal{A}}(\mathcal{M})$ will never generate an output $\perp$).

However, remember that the whole motivation for using a mapper is that this will allow us to construct a correct model for $\mathcal{M}$. Since $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$, we have accomplished our mission and may thus abort the learning process.

A second approach for constructing an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$, which we used in the experiments described in Section 3.4, consists of forwarding the test sequences for $\mathcal{H}$ computed by LearnLib to the mapper, which then forwards the concretized version of this sequence to the implementation of $\mathcal{M}$, and returns the abstracted version of the output of $\mathcal{M}$ to the *learner*. If, for all test sequences, the produced output agrees with the output predicted by the hypothesis then we consider the hypothesis to be correct, otherwise we have obtained a counterexample. In this approach, a key issue is how the mapper selects a concrete input symbol for a given abstract input symbol. In our experiments we used randomization (more specifically, a uniform distribution over the possible concrete inputs). Although the initial results in our experiments were very positive, more research will be required to find out under which conditions this is a good "approximation" of an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Given that we have an implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$, assuming that $\alpha_{\mathcal{A}}(\mathcal{M})$ is finite and behavior deterministic, and assuming that the oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ behaves correctly, LearnLib should succeed in inferring a deterministic Mealy machine $\mathcal{H}$ that is behaviorally equivalent to $\alpha_{\mathcal{A}}(\mathcal{M})$. It then follows by Theorem 3.1 that $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.

3.2 Symbolic Mealy Machines and Mappers

Even though our general approach for using abstraction in automata learning is phrased most naturally at the semantic level, an actual implementation of our approach requires a syntactic (symbolic) representation of Mealy machines and abstractions. Therefore, in this section, we present a general syntax for symbolic representation of Mealy machines and mappers.

We assume a language with (typed) variables, function, predicate, and constant symbols. We assume that each variable v comes equipped with a type $\mathsf{type}(v)$, which is the (nonempty) set of values that it may take. We postulate that for each variable v there is a primed version v', which has the same type. If V is a set of variables then we write V' to denote the set $\{v' \mid v \in V\}$. We assume that, using the variables, function, predicate, and constant symbols, it is possible to construct terms and formulas. Each term t has an associated type $\mathsf{type}(t)$. We use $\equiv$ to denote syntactic equality of terms. If V is a set of variables, then a *valuation* for V is a function that maps each variable in V to an element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for V. If ξ is a valuation for V and φ is a formula with (free) variables in V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. Similarly, if t is a term then we write $[\![t]\!]_\xi$ for the value in $\mathsf{type}(t)$ to which t evaluates under valuation ξ. If $V' \subseteq V$ then $\xi\lceil V'$ denotes the restriction of ξ to the variables in V'. If $v_1, \ldots, v_n$ are variables in V and $t_1, \ldots, t_n$ are terms with matching types, then we write $\xi[v_1, \ldots, v_n := t_1, \ldots, t_n]$ for the valuation in which all variables have the same values as in ξ except for $v_1, \ldots, v_n$ which are evaluated to $[\![t_1]\!]_\xi, \ldots, [\![t_n]\!]_\xi$, respectively.

3.2.1 Symbolic Mealy Machines

We employ a slight variation of Jonsson's [92] approach for specification of distributed systems and define a *symbolic Mealy machine* by means of a program-like notation with guarded multiple assignments. Each assignment statement is labeled with two events which denote reception and transmission of a message.

An event signature specifies the possible interactions between a symbolic Mealy machine and its environment.

Definition 3.5 (Event signature) An *event term* is an expression of the form $\varepsilon(p_1, \ldots, p_m)$, where ε is a symbol referred to as the *event primitive*, and $p_1, \ldots p_m$ pairwise different variables referred to as *parameters*. An *event signature* Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are disjoint, finite sets of event terms. We require that the event primitives of different event terms in $T_I \cup T_O$ are distinct.

Using event signatures, we can define the notion of a symbolic Mealy machine.

Definition 3.6 (Symbolic Mealy machine) A *symbolic Mealy machine (SM)* is a tuple $\mathcal{SM} = \langle \Sigma, V, \Theta, \Delta \rangle$, where

- $\Sigma = \langle T_I, T_O \rangle$ is an event signature,
- V is a finite set of variables, referred to as *state variables*, disjoint from the set of parameters of Σ,
- Θ is a formula, the *initial condition*, with (free) variables in V. We require that there is a unique valuation $q_0 \in \mathsf{Val}(V)$ such that $q_0 \models \Theta$, and
- Δ is a finite set of *transitions* of the form

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \ldots, p_m)\ \textbf{\textit{when}}\ \varphi\ \textbf{\textit{event}}\ \varepsilon_O(p_{m+1}, \ldots, p_l),$$

  where $\varepsilon_I(p_1, \ldots, p_m) \in T_I$, $\varepsilon_O(p_{m+1}, \ldots, p_l) \in T_O$, $\{p_1, \ldots, p_m\} \cap \{p_{m+1}, \ldots, p_l\} = \emptyset$, and φ is a formula with (free) variables in $\{p_1, \ldots, p_l\} \cup V \cup V'$. We require that $\mathcal{SM}$ is input enabled. Formally, if there are k transitions with event primitives $\varepsilon_I(p_1, \ldots, p_m)$ and $\varepsilon_O(p_{m+1}, \ldots, p_l)$, with formulas $\varphi_1, \ldots, \varphi_k$, respectively, and $V = \{v_1, \ldots v_n\}$, then the formula

$$\exists v_1', \ldots, v_n'\ \exists p_{m+1}, \ldots, p_l : \varphi_1 \vee \cdots \vee \varphi_k$$

  should evaluate to true.

Example 3.8 (Symbolic representation of $\mathcal{M}_{COM}$) The Mealy machine $\mathcal{M}_{COM}$ of our running Example 3.1 can be described as an SM $\mathcal{SM}_{COM} = \langle \Sigma, V, \Theta, \Delta \rangle$, where

- $\Sigma = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{\mathsf{OK}, \mathsf{NOK}\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
- $V = \{\mathsf{ID}, \mathsf{SN}, \mathsf{INIT}\}$, where ID and SN have type $\mathbb{N}$ and INIT has type $\mathbb{B}$. Variable ID stores the current session identifier, SN stores the current sequence number, and INIT records whether a session has been initialized.

- Initially, ID and SN are 0 and no session has been initialized:

$$\Theta \quad \equiv \quad \mathsf{ID} = 0 \wedge \mathsf{SN} = 0 \wedge \neg \mathsf{INIT}.$$

- Set Δ contains three transitions:

$$\begin{aligned}
&\textbf{event } REQ(\mathsf{p}_1, \mathsf{p}_2) \textbf{ when } \neg\mathsf{INIT} \wedge \mathsf{ID}' = \mathsf{p}_1 \wedge \mathsf{SN}' = \mathsf{p}_2 \wedge \mathsf{INIT}' \\
&\qquad\qquad \textbf{event } \mathsf{OK} \\
&\textbf{event } REQ(\mathsf{p}_1, \mathsf{p}_2) \textbf{ when } \mathsf{INIT} \wedge \mathsf{p}_1 = \mathsf{ID} \wedge \mathsf{p}_2 = \mathsf{SN} + 1 \wedge \\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{p}_2 \wedge \mathsf{INIT}' \textbf{ event } \mathsf{OK} \\
&\textbf{event } REQ(\mathsf{p}_1, \mathsf{p}_2) \textbf{ when } \mathsf{INIT} \wedge (\mathsf{p}_1 \neq \mathsf{ID} \vee \mathsf{p}_2 \neq \mathsf{SN} + 1) \wedge \\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{SN} \wedge \mathsf{INIT}' \textbf{ event } \mathsf{NOK}
\end{aligned}$$

  Every transition contains an input event, an output event, and a **when** clause that determines the conditions that need to hold in the current and the next state. For example, in the first transition a $REQ(\mathsf{p}_1, \mathsf{p}_2)$ input triggers an OK output whenever the session needs to be initialized. In the next state, the initialization is completed by assigning to ID the value of p_1 and to SN the value of p_2. □

The semantics of symbolic Mealy machines is defined, in a straightforward manner, in terms of Mealy machines.

Definition 3.7 (Semantics of symbolic MM) The semantics of an event term $\varepsilon(p_1, \ldots, p_m)$ is the set

$$[\![\varepsilon(p_1, \ldots, p_m)]\!] \;=\; \{\varepsilon(d_1, \ldots, d_m) \mid d_1 \in \mathsf{type}(p_1), \ldots, d_m \in \mathsf{type}(p_m)\}.$$

The semantics of a set T of event terms is defined by pointwise extension:

$$[\![T]\!] \;=\; \bigcup_{\varepsilon(p_1,\ldots,p_m) \in T} [\![\varepsilon(p_1, \ldots, p_m)]\!].$$

Let $\mathcal{SM} = \langle \Sigma, V, \Theta, \Delta \rangle$ be a symbolic Mealy machine with $\Sigma = \langle T_I, T_O \rangle$. The semantics of $\mathcal{SM}$, notation $[\![\mathcal{SM}]\!]$, is the Mealy machine $\langle I, O, Q, q_0, \rightarrow \rangle$ where

- $I = [\![T_I]\!]$,
- $O = [\![T_O]\!]$,
- $Q = \mathsf{Val}(V)$,
- $q_0 \in \mathsf{Val}(V)$ is the unique valuation satisfying $q_0 \models \Theta$, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the smallest relation that satisfies

$$\frac{\begin{array}{c}(\textbf{\textit{event}}\ \varepsilon_I(p_1, \ldots, p_m)\ \textbf{\textit{when}}\ \varphi\ \textbf{\textit{event}}\ \varepsilon_O(p_{m+1}, \ldots, p_l)) \in \Delta \\ \forall j \leq l, \xi(p_j) = d_j \\ \forall v \in V, \xi(v) = q(v) \text{ and } \xi(v') = q'(v) \\ \xi \models \varphi\end{array}}{q \xrightarrow{\varepsilon_I(d_1,\ldots,d_m)/\varepsilon_O(d_{m+1},\ldots,d_l)} q'}$$

The reader may check that the semantics of the symbolic Mealy machine $\mathcal{SM}_{COM}$ described in Example 3.8 indeed yields the Mealy machine $\mathcal{M}_{COM}$ of Example 3.1: the only difference is that states (id, sn, b) of $\mathcal{M}_{COM}$ correspond to valuations in $\mathcal{SM}_{COM}$, in which variable ID has value id, variable SN has value sn, and variable INIT has value b. In this chapter, we only consider symbolic Mealy machines whose semantics is input enabled, as required for a Mealy machine.

In the same way as symbolic Mealy machines constitute a syntactic representation of Mealy machines, the definition below introduces *symbolic mappers* as a syntactic representation of mappers. The abstract event signature of a symbolic mapper is the same as its concrete event signature, except that the parameters have a different (typically smaller) domain.

Definition 3.8 (Symbolic mapper) Let $\Sigma_c = \langle T_I, T_O \rangle$ be an event signature. A *symbolic mapper* for Σ_c is a structure $\mathcal{SA} = \langle \Sigma_c, \Sigma_a, V, \Theta, \Delta, \Psi \rangle$, where

- $\Sigma_a = \langle T_X, T_Y \rangle$ is an event signature, referred to as the *abstract event signature*. We require that, for each $\varepsilon(p_1, \ldots, p_m) \in T_I$, T_X contains an element $\varepsilon(q_1, \ldots, q_m)$. Similarly, we require that, for each $\varepsilon(p_1, \ldots, p_m) \in T_O$, T_Y contains a corresponding element $\varepsilon(q_1, \ldots, q_m)$,
- $V = \{v_1, \ldots, v_n\}$ is a finite set of variables, disjoint from the set of parameters of Σ_c,
- Θ is a formula, the *initial condition*, whose free variables are in V. We require that there exists a unique valuation $r_0 \in \mathsf{Val}(V)$ such that $r_0 \models \Theta$,
- Δ is a finite set of *transitions* given by

  $$\textbf{\textit{event}}\ \varepsilon(p_1, \ldots, p_m)\ \textbf{\textit{when}}\ \varphi\ \textbf{\textit{do}}\ \langle v_1, \ldots, v_n \rangle := \langle t_1, \ldots, t_n \rangle,$$

  where $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, φ is the *guard*, a formula with variables in $V \cup \{p_1, \ldots, p_m\}$, and $t_1, \ldots, t_n$ are terms with variables in $V \cup \{p_1, \ldots, p_m\}$. We require that $\mathcal{SA}$ is input and output enabled: for each $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, the disjunction of the set of guards of transitions for that event primitive is equivalent to true. Furthermore, we require that $\mathcal{SA}$ is deterministic: whenever we have two different transitions for the same event primitive then the conjunction of their guards is equivalent to false, and
- Ψ is a finite set of *event abstractions* which contains, for each event term $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, an expression $\varepsilon(e_1, \ldots, e_m)$, where, for each j, e_j is a term with variables in $V \cup \{p_1, \ldots, p_m\}$ and $\mathsf{type}(e_j) = \mathsf{type}(q_j)$.

Example 3.9 (Symbolic mapper) We illustrate how the mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$, which we defined in Example 3.2, can also be described as a symbolic mapper $\mathcal{A}_S$:

- $\Sigma_c = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{\mathsf{OK}, \mathsf{NOK}\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
  Note that Σ_c equals the event signature Σ of Example 3.8.
- $V = \{curId, curSn\}$, where $curId$ and $curSn$ are variables of type $\mathbb{N} \cup \{\bot\}$, used to store the identifier and sequence number, respectively, of the current session.

- In the initial state both variables are undefined:

$$\Theta \quad \equiv \quad curId = \bot \wedge curSn = \bot \ .$$

- Set Δ contains five transitions:

event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId = \bot$
	do $\langle curId, curSn\rangle := \langle \mathsf{p}_1, \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge \mathsf{p}_1 = curId \wedge \mathsf{p}_2 = curSn + 1$
	do $\langle curSn\rangle := \langle \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge (\mathsf{p}_1 \neq curId \vee \mathsf{p}_2 \neq curSn + 1)$
	do $\langle\rangle := \langle\rangle$
event OK	**when** TRUE **do** $\langle\rangle := \langle\rangle$
event NOK	**when** TRUE **do** $\langle\rangle := \langle\rangle$

  The first transition, for instance, states that when receiving a *REQ* input and variable *curId* still has its initial value, we need to assign the state variables the values received in the input message.

- $\Sigma_a = \langle\{REQ(\mathsf{q}_1, \mathsf{q}_2)\}, \{\mathsf{OK}, \mathsf{NOK}\}\rangle$, where *REQ*, OK and NOK are event primitives and q_1 and q_2 are parameters of type {CUR, OTHER}.

- Set Ψ contains three event abstractions:
  - $REQ($ `if` $curId = \bot \vee curId = \mathsf{p}_1$ `then` CUR `else` OTHER , `if` $curSn = \bot \vee curSn + 1 = \mathsf{p}_2$ `then` CUR `else` OTHER $)$
  - OK
  - NOK □

The semantics of symbolic mappers is defined, again in a straightforward manner, in terms of mappers.

Definition 3.9 (Semantics of symbolic mapper) Let $\mathcal{SA} = \langle \Sigma_c, \Sigma_a, V, \Theta, \Delta, \Psi\rangle$ be a symbolic mapper for Σ_c. Let $\Sigma_a = \langle T_X, T_Y\rangle$. The semantics of $\mathcal{SA}$, notation $[\![\mathcal{SA}]\!]$, is the mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, abstr\rangle$, where

- $I = [\![T_I]\!]$,
- $O = [\![T_O]\!]$,
- $X = [\![T_X]\!]$,
- $Y = [\![T_Y]\!]$,
- $R = \mathsf{Val}(V)$,
- r_0 is the unique valuation satisfying $r_0 \models \Theta$,

- $\delta(r, \varepsilon(d_1, \ldots, d_m)) = r'$, where r' is given by the rule

$$\frac{\begin{array}{c}(\textbf{\textit{event}}\ \varepsilon(p_1, \ldots, p_m)\ \textbf{\textit{when}}\ \varphi\ \textbf{\textit{do}}\ \langle v_1, \ldots, v_n\rangle := \langle t_1, \ldots, t_n\rangle) \in \Delta \\ \forall j \leq m, \xi(p_j) = d_j \qquad r \cup \xi \models \varphi \\ r' = ((r \cup \xi)[v_1, \ldots, v_n := t_1, \ldots, t_n])\lceil V\end{array}}{r \xrightarrow{\varepsilon(d_1,\ldots,d_m)} r'}$$

- for all $r \in R$, $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, ξ a valuation of $\{p_1, \ldots, p_m\}$ such that, for $1 \leq j \leq m$, $\xi(p_j) = d_j$, and $\varepsilon(e_1, \ldots, e_m) \in \Psi$,

$$abstr(r, \varepsilon(d_1, \ldots, d_m)) = \varepsilon([\![e_1]\!]_{r\cup\xi}, \ldots, [\![e_m]\!]_{r\cup\xi}).$$

Given a symbolic mapper $\mathcal{SA}$ and a Mealy machine (hypothesis) $\mathcal{H}$, we may construct a symbolic Mealy machine $\gamma^S_{\mathcal{SA}}(\mathcal{H})$ such that $[\![\gamma^S_{\mathcal{SA}}(\mathcal{H})]\!]$ is isomorphic to $\gamma_{[\![\mathcal{SA}]\!]}(\mathcal{H})$. Since the construction is routine and not required for the remainder of this chapter, we leave it to the reader to work out the details.

3.3 Systematic Construction of Abstractions

The construction of a suitable mapper component is an important part of our technique for generating a model of an SM $\mathcal{SM}$. In general, the construction of the mapper will rely on insights into what aspects of the data parameters are important for the behavior of $\mathcal{SM}$. But it is also possible to present guidelines for constructing them systematically, from which also automated support can be developed. In this section, we suggest a set of such guidelines.

To simplify our presentation, we assume that output event primitives do not have parameters, as is the case, e.g., in Example 3.9. Then the main purpose of the mapper is to provide an abstraction of the parameters of input symbols, which preserves the information that determines which output symbols will subsequently be generated in an observation. More precisely if (u, s) and (u', s') are different observations of $\mathcal{SM}$, which the mapper abstracts to $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$, then $S \neq S'$ should imply $U \neq U'$, otherwise the abstraction $\alpha_{\mathcal{A}}(\mathcal{SM})$ will behave nondeterministically, something that the learning algorithm is not designed for. The requirement to produce a behavior deterministic abstraction suggests a methodology for constructing mappers, in which observed nondeterminism in $\alpha_{\mathcal{A}}(\mathcal{SM})$ triggers a modification of the mapper. The modification of the mapper can be performed on the fly: Whenever nondeterminism is observed during learning, the abstraction is refined and the entire learning process is restarted from scratch.

One can start from an initial mapper, whose event abstractions are trivial, i.e., they map any value of any parameter in any input symbol to a single abstract value. Whenever a sequence of output queries shows that the composition of mapper and $\mathcal{SM}$ is nondeterministic, i.e., there is a pair of observations, (u, s) and (u', s'), such that with $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$ we have $S \neq S'$ but $U = U'$, then some event abstraction that contributes to generating U or U' must be refined. This refinement is constructed by first performing additional output queries to determine in what way the parameters in u and u' cause S and S'

to be different. In many cases, it is possible to find a particular condition that determines whether the output will be S or S'. This condition is then introduced into the mapper in order to differentiate between $\tau_{\mathcal{A}}(u, s)$ and $\tau_{\mathcal{A}}(u', s')$. In the case that the new condition refers to parameters in different symbols of u and u', variables must be introduced into the mapper that remember received data values, in order that the new condition can refer to them.

Let us illustrate how these guidelines can be applied in Example 3.9. We start from an initial (too coarse) abstraction, in which the mapper does not distinguish between different parameter values in input symbols of form $REQ(d_1, d_2)$. By performing output queries, we discover that the resulting composition of mapper and $\mathcal{SM}$ is nondeterministic. Namely, an input of form $REQ(d_1, d_2)$ may give rise either to an output OK or an output NOK. Additional investigation by means of output queries, in order to find a distinction between these two cases, reveals that the OK output occurs precisely in the case that

- d_1 occurred in the first input of form $REQ(d_1', d_2')$ (with $d_1' = d_1$), and
- $d_2 - 1$ occurred in the most recent input of form $REQ(d_1', d_2')$, which resulted in an OK response from $\mathcal{SM}$.

As a result of these insights, we let the mapper have

- one variable (say, *curId*) which stores the value of d_1' in the first input of form $REQ(d_1', d_2')$, and
- one variable (say, *curSn*) which stores the value of d_2' whenever an input of form $REQ(d_1', d_2')$ arrives and results in an OK response.

Furthermore, we refine the event abstraction for $REQ(\mathsf{p}_1, \mathsf{p}_2)$, as follows.

- p_1 is mapped to one abstract value (say, CUR) if its value is equal to the value of *curId*, and to another value (say, OTHER) otherwise.
- p_2 is mapped to one abstract value (say, CUR) if its value is equal to the value of $curSn + 1$, and to another value (say, OTHER) otherwise.

By completing the mapper based on this abstraction, e.g., also investigating how to handle initialization of variables, we obtain the mapper that is presented in Example 3.9.

In Chapter 6, we show how, following the approach sketched above, mappers can be constructed fully automatically for a restricted class of symbolic Mealy machines in which one can test for equality of data parameters, but no operations on data are allowed.

3.4 Experiments

We implemented and applied our approach to infer models of two implemented standard protocols: the session initiation protocol (SIP) and the transmission control protocol (TCP). As *learner*, we used an efficient implementation of the L^* algorithm in LearnLib [131, 114]. LearnLib also provides several implementations

of model-based test algorithms in order to realize equivalence queries, including random test suites of user-controlled size. Hence, in our experiments, the *teacher* consisted of an SUT, which is a protocol implementation, in combination with a model-based test algorithm implemented in LearnLib. We postulate that the behavior of the SUT can be modeled as a Mealy machine (cf. the notion of *test hypothesis* from model-based testing [151]) and our task is to learn this unknown Mealy machine.

3.4.1 The Session Initiation Protocol (SIP)

The session initiation protocol (SIP) is an application layer protocol for creating and managing multimedia communication sessions [136]. Although extensive documentation is available, there is no reference model in the form of a state machine. We aimed to infer the behavior of a SIP Server entity when setting up connections with a SIP Client. As system under test (SUT) we used an implementation of SIP in the protocol simulator ns-2 [123], Messages were represented as C++ structures, saving us the trouble of parsing messages represented as bitstrings. The set of messages that can be exchanged between a SIP Client and a SIP Server can be described by the event signature $\Sigma_{SIP} = \langle T_I, T_O \rangle$. Set T_I contains event terms of the form $Method(CallId, CSeq, Via)$, where $Method = \{\text{INVITE}, \text{PRACK}, \text{ACK}\}$ is the set of input event primitives, which correspond to the different types of requests that can be made by the client:

- an INVITE request is an initial request needed for session establishment. It indicates that a SIP Client wants to establish a connection with the SIP Server. This activity can be compared with dialing someone's telephone number.
- a PRACK request is an acknowledgement, which is used to confirm provisional responses that could have been lost otherwise.
- an ACK request confirms that a Client has received a final response to an INVITE request. Unlike PRACK, an ACK request does not have a response.

Set T_O contains event terms of the form $StatusCode(CallId, CSeq, Via)$, where $StatusCode = \{100, 180, 183, 200, 481, 486\}$ is the set of output event primitives. The three digit status codes indicate the outcome generated by the Server in response to a previous request by the Client:

- *1xx* responses are provisional responses. A provisional response is sent when the associated request was received but the request still needs to be processed. Possible *1xx* responses are *100* (Trying), *180* (Ringing), which means that the recipient's phone is ringing, and *183* (Session Progress),
- *2xx* responses are positive final responses. They indicate that the request was successful. A *200* (OK) response is sent when a user accepts invitation to a session, and
- *4xx* responses are negative final responses. They indicate that the request contains bad syntax or cannot be fulfilled at the Server. Possible *4xx* responses are *481* (Call/Transaction Does Not Exist) and *486* (Busy Here).

A typical interaction between a Client and a Server is visualized in Table 3.1.

Client	**Server**
INVITE(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) →	
	← 100(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) Trying
	← 183(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) Session Progress
PRACK(*CallId*:4, *CSeq*:2, *Via*:1.1.2;branch=z9hG4bK3) →	
	← 200(*CallId*:4, *CSeq*:2, *Via*:1.1.2;branch=z9hG4bK3) OK
ACK(*CallId*:4, *CSeq*:1, *Via*:1.1.2;branch=z9hG4bK3) →	

Table 3.1: *Typical session establishment in SIP*

All of the above event terms have the same parameters:

- *CallId* is a unique session identifier,
- *CSeq* is a sequence number that orders transactions in a session, and
- *Via* specifies the transport path that is used for the transaction. The *Via* parameter is a pair, consisting of a default address and a variable branch.

The actual messages that are used within SIP carry some additional parameters, specifying the addresses of the originator and receiver of a request, and the address where the Client wants to receive input messages. These parameters must be preconfigured in a session with ns-2, so they are set to constant values throughout the experiment, and play no role in the learning. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed.

The SUT does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, set T_I contains an additional event term NIL(), which denotes the absence of input (in order to allow sequences of outputs), and set T_O contains an additional event term TIMEOUT(), denoting the absence of output.

Following Definition 3.8, a symbolic mapper $\mathcal{SA}$ for SIP can be defined as follows. Monitoring of output queries, as described in Section 3.3 reveals that for each of these parameters, the ns-2 SIP implementation remembers the value which is received in the first INVITE message (presumably, it is interpreted as parameters of the connection that is being established), and also the value received in the most recent input message when producing the corresponding reply. We therefore equip the mapper with six state variables. Variable *firstInviteId* stores the

CallId parameter of the first *Invite* message, and variable *lastId* stores the *CallId* parameter value of the most recently received message. Variables *firstInviteCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the variables *firstInviteVia* and *lastVia* for the *Via* parameter. Initially, all six variables have the undefined value $\perp$. Note that we have to remember these six state variables, because all of them are employed to construct the correct reply, i.e., they are needed to map a concrete output message to an abstract output message.

The transitions define when which state variables have to be updated, e.g.

$$\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\ \textbf{when}\ \mathit{firstInviteId} = \perp$$

$$\textbf{do}\ \langle \mathit{firstInviteId}, \mathit{firstInviteCSeq}, \mathit{firstInviteVia}\rangle := \langle CallId, CSeq, Via\rangle;$$

$$\langle \mathit{lastId}, \mathit{lastCSeq}, \mathit{lastVia}\rangle := \langle CallId, CSeq, Via\rangle$$

states that when receiving an INVITE input and the *firstInviteId* state variables still has its initial value, the mapper needs to assign the *firstInvite* and *last* state variables the values received in the input message. If the *firstInviteId* state variable does not have its initial value when an INVITE input is received, the following transition occurs:

$$\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\ \textbf{when}\ \mathit{firstInviteId} \neq \perp$$

$$\textbf{do}\ \langle \mathit{lastId}, \mathit{lastCSeq}, \mathit{lastVia}\rangle := \langle CallId, CSeq, Via\rangle$$

For PRACK and ACK inputs, the update of state variables is defined by

$$\textbf{event}\ \mathit{(PR)ACK}(CallId, CSeq, Via)\ \textbf{when}\ \mathsf{TRUE}$$

$$\textbf{do}\ \langle \mathit{lastId}, \mathit{lastCSeq}, \mathit{lastVia}\rangle := \langle CallId, CSeq, Via\rangle$$

For event term NIL() and all output event terms no state variables are updated, and we have trivial transitions of the form **when** TRUE **do** $\langle\rangle := \langle\rangle$.

Additional monitoring of output queries reveals that the mapper needs to consider two cases in the abstraction of the *CallId* parameter in input messages:

1. The concrete value of *CallId* is a fresh value or equal to the *firstInviteId* state variable. In this case *CallId* should be mapped to the abstract value FIRST.

2. The concrete value of *CallId* is NOT a fresh value and NOT equal to the *firstInviteId* state variable. In this case, *CallId* should be mapped to the abstract value LAST.

Both events require the use of the *firstInviteId* state variable. We define the relation between concrete and abstract input symbols by the event abstractions

$$Method(\ \begin{array}{c}\texttt{if}\ (\mathit{firstInviteId} = \perp \vee \mathit{firstInviteId} = CallId)\\ \texttt{then}\ \text{FIRST}\ \texttt{else}\ \text{LAST}\end{array}\ , \text{ANY}, \text{ANY}),$$

where *Method* can be any input event primitive. The input parameters *Via* and *CSeq* are always mapped to the abstract value ANY, since we found that ns-2 always behaves in the same way - no matter which concrete value has been selected.

To cope with unexpected values that might be returned by the SUT, different from the values recorded in the state variables, we added an abstract value OTHER. We define the relation between concrete and abstract output symbols by the event abstraction $StatusCode(e_1, e_2, e_3)$, where $StatusCode$ can be any output event primitive,

$$
\begin{aligned}
e_1 \;=\;& \texttt{if } CallId = \mathit{firstInviteId} \texttt{ then FIRST} \\
& \quad \texttt{elseif } CallId = \mathit{lastId} \texttt{ then LAST else OTHER,} \\
e_2 \;=\;& \texttt{if } CSeq = \mathit{firstInviteCSeq} \texttt{ then FIRST} \\
& \quad \texttt{elseif } CSeq = \mathit{lastCSeq} \texttt{ then LAST else OTHER, and} \\
e_3 \;=\;& \texttt{if } Via = \mathit{firstInviteVia} \texttt{ then FIRST} \\
& \quad \texttt{elseif } Via = \mathit{lastVia} \texttt{ then LAST else OTHER.}
\end{aligned}
$$

Since event terms NIL() and TIMEOUT() carry no parameters, the event abstraction for these terms is trivial.

Results The inference performed by LearnLib needed about one thousand output queries (sequences of inputs) and one equivalence query, took about one hour, and resulted in an abstract model $\mathcal{H}$ with 9 locations and 63 transitions. This model can be found in Appendix 3.B. For presentation purposes, we have also included a simplified version of model $\mathcal{H}$ in Appendix 3.A. In this pruned model, we removed transitions with output symbol $\perp$ and transitions with an empty input and output symbol, i.e., *NIL/TIMEOUT*. In Appendix 3.A, we show the pruned abstract model with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter of the input messages with abstract value FIRST is depicted in the model with solid transition lines, the remaining transitions have a dashed line pattern. We suppressed all other parameters in the figure.

The abstract model $\mathcal{H}$ does not contain any output parameter value OTHER: apparently all concrete output values generated by the SUT are mapped to either FIRST or LAST. This implies that mapper $[\![\mathcal{SA}]\!]$ is output-predicting for all the outputs that occur in transitions of $\mathcal{H}$, since the abstract values FIRST and LAST always correspond to a single concrete value. Hence, by Lemma 3.4, Mealy machine $[\![\gamma^S_{\mathcal{SA}}(\mathcal{H})]\!]$ is deterministic. If $\mathcal{M}$ is a Mealy machine that models the SUT then, according to Lemma 2.1, $\mathcal{M} \approx [\![\gamma^S_{\mathcal{SA}}(\mathcal{H})]\!]$. Thus, using our approach, we have succeeded to learn a model that is observation equivalent to the (unknown) model $\mathcal{M}$ of the ns-2 implementation of the SIP protocol.

3.4.2 The Transmission Control Protocol (TCP)[1]

As a second case study, we studied the implementation of the transmission control protocol (TCP) [126, 148] in Windows 8. TCP is a transport layer protocol,

[1]Paul Fiterău-Broştean helped us with the TCP experiments, using the setup developed by Ramon Jansen in his bachelor's thesis [91].

which provides reliable and ordered delivery of a byte stream from one computer application to another. It is one of the core protocols of the Internet Protocol Suite. We considered the connection establishment and closing between a Client and a Server, but left out the data transfer phase. As *SUT*, we consider an implementation of the Server component of the protocol in Windows 8. Our setup was restrictive, in that we did not explicitly use triggers, like CONNECT, SEND, LISTEN, and CLOSE. Thus our learned model reflects only setup and closing of connections that are initiated by the Client communicating with the Server. To include setup and closing initiated by the learned Server, we should also have included the above triggers in the set of input symbols of output queries.

For our experiments we used virtualization through Virtual Box. LearnLib, a Java implementation of the mapper and an adapter were deployed on a guest Ubuntu 12.04 LTS operating system, while the server was deployed on a host Windows 8 machine. A Python adapter based on Scapy was used to construct and send request packets and retrieve response packets. In our experiments, we considered *Request* messages, which are input to the *SUT*, and *Response* messages, which are output by the *SUT*. We ignored a number of fields in TCP messages (these are kept to a constant value in our learning experiments) and consider messages of the form *Request*/*Response*(*Flag*, *SeqNr*, *AckNr*). Parameter *Flag* consists of four bits *SYN*, *ACK*, *FIN* and *RST* that define what type of message is sent: *SYN* synchronizes sequences numbers, *ACK* acknowledges the previously received sequence number, *FIN* signals the end of the data transfer phase, and *RST* resets the protocol to its initial state. Table 3.2 lists the seven possible values for parameter *Flag* that we considered in our experiments.[2] We write *RST*(*Flag*) as shorthand for $Flag \in \{\text{RST}, \text{RST+ACK}\}$. Notation *ACK*(*Flag*) is defined similarly. Parameter

Flag	*SYN*	*ACK*	*FIN*	*RST*
SYN	1	0	0	0
SYN+ACK	1	1	0	0
ACK	0	1	0	0
FIN	0	0	1	0
FIN+ACK	0	1	1	0
RST	0	0	0	1
RST+ACK	0	1	0	1

Table 3.2: *Possible values for Flag parameter.*

SeqNr is a sequence number that needs to be synchronized with both sides of the connection, and parameter *AckNr* acknowledges a previous sequence number. TCP sequence and acknowledgement numbers have 32 bits and thus the values of *SeqNr* and *AckNr* are contained in the interval $[0, 2^{32} - 1]$. In addition, there is an output symbol *timeout*, which corresponds to the scenario in which the Server does not generate any response.

To define the mapper, we use information obtained from the standard [126]. The mapper uses variables *lastSeqSent* and *lastAckSent* to record the last se-

[2] Uijen [156] also describes a more general learning experiment in which all possible combinations of the control bits *SYN*, *ACK* and *FIN* are allowed, including the so-called Kamikaze packet [126] in which all the flag bits are turned on.

quence number and acknowledgement, respectively, that have been transmitted to the *SUT* in a valid request message. Similarly, the mapper uses variables *lastSeqReceived* and *lastAckReceived* to record the last sequence number and acknowledgement, respectively, that have been received from the *SUT* in a valid response message. The variables *lastSeqSent*, *lastAckSent*, *lastSeqReceived* and *lastAckReceived* all have domain $[0, 2^{32} - 1]$ and initial value 0. Boolean variable *INIT*, which is TRUE initially, is used to record whether client and host have already exchanged sequence and acknowledgment numbers in the current session. We formally define the update function of the mapper by the following three transitions:

event *Request*(*Flag*, *SeqNr*, *AckNr*) **when**	TRUE **do**
$\langle lastSeqSent, lastAckSent, INIT\rangle :=$	$\langle SeqNr, AckNr, INIT \vee RST(Flag)\rangle$
event *Response*(*Flag*, *SeqNr*, *AckNr*) **when**	TRUE **do**
$\langle lastSeqReceived, lastAckReceived, INIT\rangle :=$	$\langle SeqNr, AckNr, RST(Flag)\rangle$
event *timeout*() **when**	TRUE **do** $\langle\rangle := \langle\rangle$

Our abstraction function partitions parameter values into two classes: valid and invalid. The sequence number of the first request is always valid. The sequence number of a subsequent request is valid if it equals the last acknowledgement that has been received. The acknowledgement number of the initial request is always valid. An acknowledgement is also always valid when the *ACK* bit is 0. For the remaining requests the acknowledgement is valid when it is obtained by incrementing (modulo 2^{32}) the last sequence number that has been received:

$$\begin{aligned} ValidReqSeq &\equiv INIT \vee SeqNr = lastAckReceived \\ ValidReqAck &\equiv INIT \vee AckNr = lastSeqReceived + 1 \end{aligned}$$

Validity of response messages is defined similarly. In the initial state, we cannot predict the sequence number of an incoming response message. In all the other states there is a unique valid sequence number. In all states, including the initial one, we are able to predict the acknowledgement number of an incoming response message.

$$\begin{aligned} ValidResSeq &\equiv INIT \vee SeqNr = \textbf{if } (Flag = \text{RST+ACK}) \textbf{ then } 0 \\ &\quad\ \textbf{else } lastAckSent \textbf{ fi} \\ ValidResAck &\equiv (ACK(Flag) \wedge AckNr = lastSeqSent + 1) \vee \\ &\quad\ (Flag = RST \wedge AckNr = lastAckSent) \end{aligned}$$

The relation between concrete and abstract symbols is concisely specified by following three event abstractions:

timeout()
Request(*Flag*, *ValidReqSeq*, *ValidReqAck*)
Response(*Flag*, *ValidResSeq*, *ValidResAck*)

Note that these abstractions preserve the value of the *Flag* parameter. Altogether we have $7 \times 2 \times 2 = 28$ abstract inputs. Our abstraction is not output-predicting since we (obviously) cannot predict the sequence number in the initial state.

Results During the learning experiments we initially only used the 7 valid inputs, following the approach of Section 2.3. After inference, LearnLib produced a model with 4 locations and $4 \times 7 = 28$ transitions. In order to display the model in this paper, we suppressed all the self-loop transitions with output *timeout*. This results in the model $\mathcal{H}$ with 4 states and 15 transitions shown in Appendix 3.C. LearnLib needed 239 membership (output) queries to learn the model, which required 92439 ms. The first hypothesis was already the final model; no refinements were needed. We tested the correctness of the learned model using the test generation algorithm of LearnLib (5000 traces with length varying from 50 to 100). These testing experiments took about 7.5 hours. If the behavior of the SUT can be described by Mealy machine $\mathcal{M}$, the above mapper is denoted by $\mathcal{A}$ and the set of valid abstract inputs by X_v, then, assuming the hypothesis model $\mathcal{H}$ is correct, $\alpha_{\mathcal{A}}(\mathcal{M}) \downarrow X_v \leq \mathcal{H}$. If X is the set of all abstract inputs then, by Lemma 2.4, $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H} \uparrow X$ and, by Theorem 3.1, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H} \uparrow X)$. Thus we may use the abstract model $\mathcal{H}$ to construct an over-approximation of the TCP host behavior. This learned model is clearly consistent with the state diagram given in the standard [126, 148]. TCP hosts just ignore incoming requests messages with invalid sequence numbers. This means it is easy to extend our learning experiments to larger alphabets with inputs *Request*(*Flag*, *ValidReqSeq*, *ValidReqAck*) satisfying *ValidReqSeq* $\Rightarrow$ *ValidReqAck*: we obtain the same model with extra self-loops for the invalid inputs. Handling request messages with valid inputs but invalid acknowledgements, and in particular defining a mapper that predicts the outputs for these messages turns out to be more involved, and is left as future work.

3.5 Conclusions and Future Work

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. We have shown the applicability of our approach for inference of (fragments of) realistic communication protocols, by feasibility studies on SIP and TCP. In Chapters 4 and 8, we describe the successful application of our approach for learning models of the Biometric passport and the Bounded Retransmission Protocol. In future work, we intend to apply our approach to larger fragments of SIP and TCP, and also to other protocols. Our work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols. Initial steps in this direction are reported in Chapter 6 and in [38, 113].

3.A Pruned SIP Model

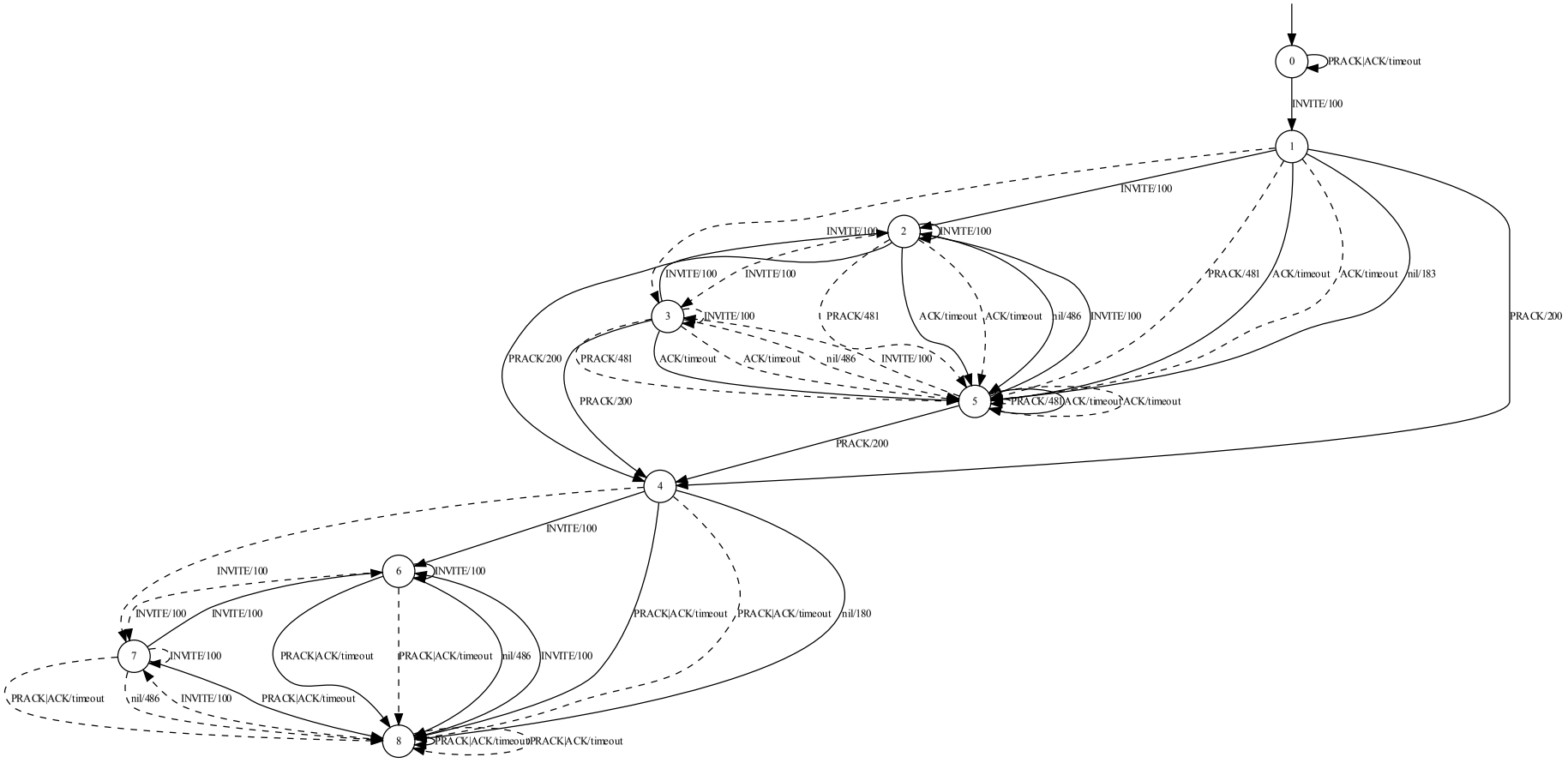


Figure 3.5: *Pruned SIP model*

3.B Complete SIP Model

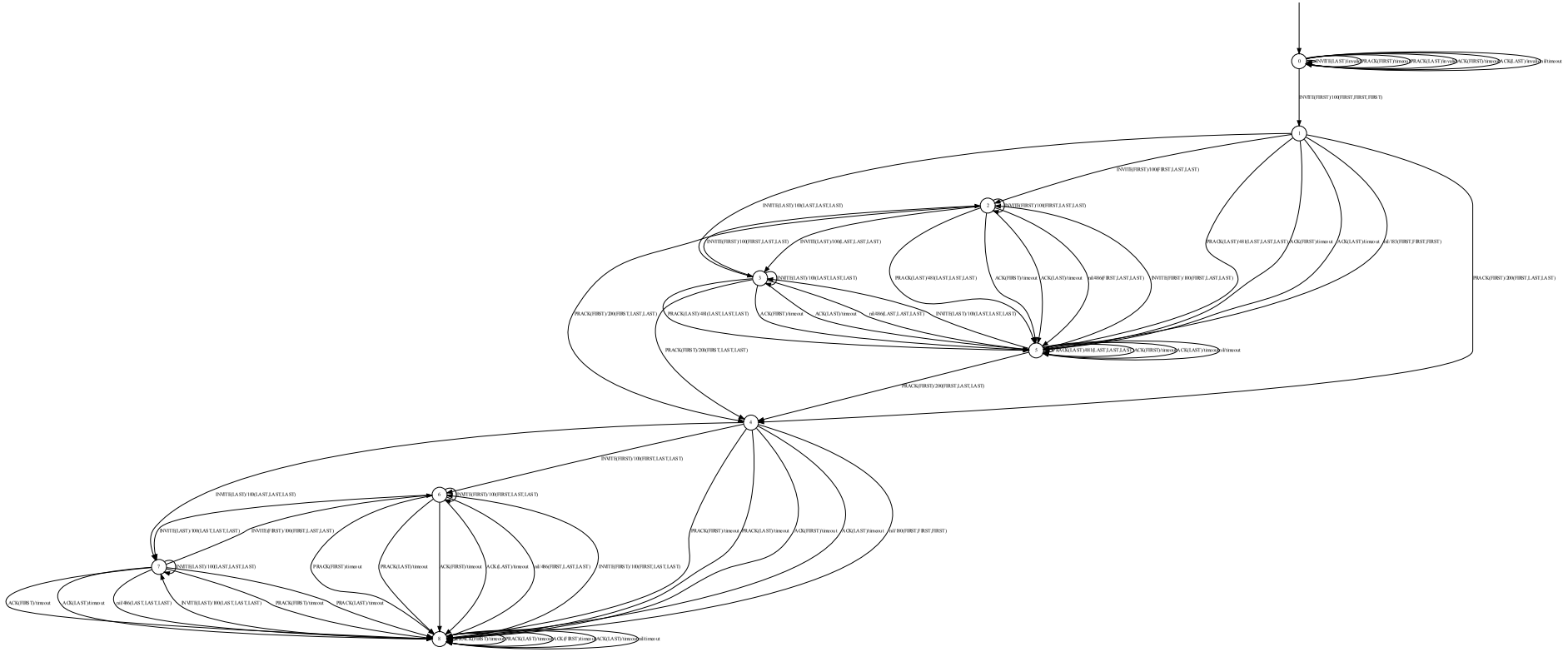


Figure 3.6: *Complete SIP model*

3.C Model of TCP Server

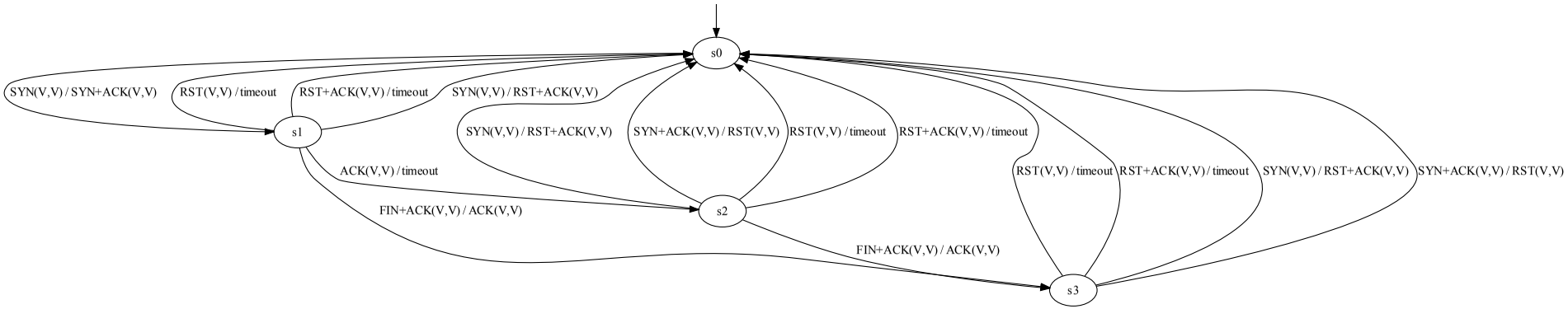


Figure 3.7: *Learned model of the TCP server. For readability, we write* $Flag(SeqNr, AckNr)/Flag'(SeqNr', AckNr')$ *instead of* $Request(Flag, SeqNr, AckNr)/\ Response(Flag', SeqNr', AckNr')$*. Moreover, V represents the VALID equivalence class. Also, we omitted all the self-loops with output 'timeout' from the diagram. There is a clear correspondence between the states of the learned model and the states mentioned in the TCP standard [126, 148]:* $s0$ *corresponds to the LISTEN state from the standard,* $s1$ *to the SYN_ RCVD state,* $s2$ *to the ESTABLISHED state, and* $s3$ *to the CLOSE_ WAIT state.*

Chapter 4
Inference and Abstraction of the Biometric Passport

In this chapter, we apply the abstraction techniques described in the previous chapter to learn a model of the new generation of biometric passports [87, 37]. This speeds up the learning process and reduces model size. In contrast to the SIP and TCP experiments in Section 3.4, we validate our automatically derived model against a previous hand-made specification of the passport [118]. This specification was used to validate the Dutch implementation of the biometric passport using the ioco-theory for MBT [152]. We implemented our abstraction as a mapping module and connected it to the LearnLib library for regular inference [130, 131, 114]. After translating our automatically derived Mealy machine to a labeled transition system (LTS), we used the tool JTorX [25] to show that this learned model is ioco-conforming to the hand-made specification. Our model can be learned within one hour and is of comparable complexity and readability as the hand-made one. It took several hours to develop the latter.

Our main contribution is to demonstrate and validate the applicability of our abstraction technique for learning automata to a practical and realistic case-study. The main result is that the model learned is comparable in size and correct w.r.t. to a previously hand-made specification. The time needed for a computer to learn the model from an existing implementation is much less than the time needed by a human to develop it.

The rest of this chapter is organized as follows. In the next section, we give an overview of our approach. Section 4.2 gives a short overview of the biometric passport; the experiments and according results are reported in Section 4.3. Finally, Section 4.4 contains conclusions and directions for future work.

4.1 Approach and Implementation

Our approach works as follows. The goal is to learn a model of an SUT - the biometric passport. For the learning process we use three components: a *learner* (LearnLib, see Section 2.2), a *teacher* (SUT), and an intermediate layer called *Abstraction mapping* that reduces the alphabet of the SUT, see Learning box in Figure

4.1. The abstraction mapping is created using a priori knowledge extracted from informal specifications, observing the behavior of the *SUT*, and several interviews with experts from our university's security department [118, 117]. Eventually, the learning algorithm generates a Mealy machine model of the *SUT*. If a reference model is available, we can validate the learned implementation to check whether it is correct with respect to the specification. In our approach, we use the testing relation **ioco** [152, 153], which is implemented in the JTorX tool [25]. The Mealy machine model has to be transformed to an input-output transition system (IOTS) to allow comparison with the specification represented as a LTS, see Validation box in Figure 4.1. We use an abstracted version of the specification to conform to the alphabet defined in the IOTS. The abstract LTS is based on a formal model created by Mostowski et al. [118] to adopt model-based testing. Their model was fed to the testing tool TorXakis (based on TorX [155]) that automatically generates and executes test cases on the fly. By comparing the responses of the *SUT* to those specified in the model, a verdict can be made, see MBT box in Figure 4.1.

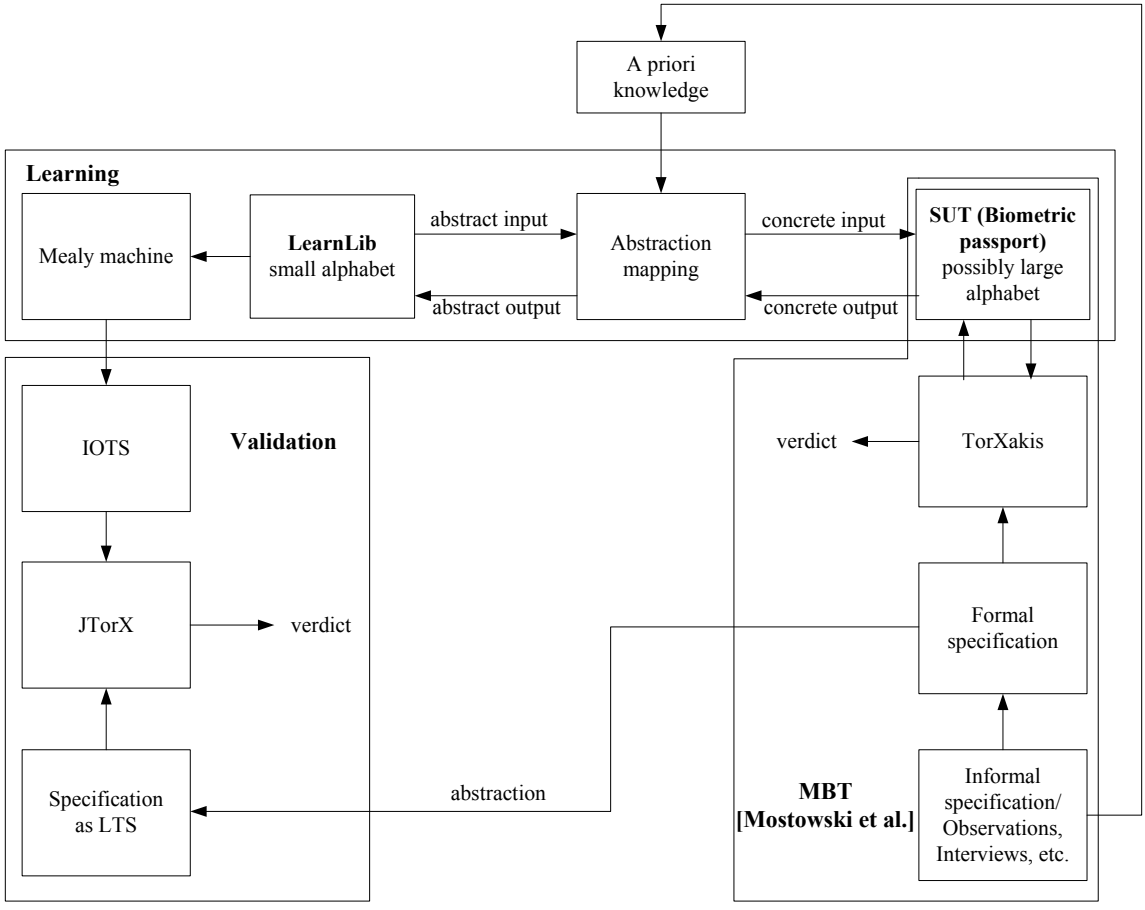


Figure 4.1: *Approach and tool implementation*

4.2 Biometric Passport

The biometric passport is an electronic passport provided with a computer chip and antenna to authenticate the identity of travelers. The data stored on the passport are highly confidential, e.g. they might contain fingerprints or an iris scan

of its owner, and are protected via several mechanisms to avoid and detect attacks. Examples of used protocols are basic access control (BAC), active authentication (AA), and extended access control (EAC) [37]. Official standards are documented in the international civil aviation organization's (ICAO) Doc 9303 [87].
In this chapter, we take a look at the interaction of the following messages:

- *Reset* resets the system.
- *GetChallenge* followed by *CompleteBAC* forms a *BAC*, which establishes secure messaging with the passport by encrypting transmitted information.
- *FailBAC* constitutes an invalid *BAC*.
- *ReadFile(int file)* tries to access highly sensitive data specified in a certain file, which is represented as an integer value in the range from 256 up to (and including) 511.
- *AA* prevents cloning of passport chips.
- *CA* followed by *TA* forms an *EAC*, which uses mutual authentication and stronger encryption than *BAC* to control access to highly confidential data.
- *FailEAC* constitutes a valid *CA* and an invalid *TA*.

For each of these messages a value *OK* or *NOK* may be returned by the *SUT*. A global overview of the behavior is depicted in Figure 4.2, where a *BAC* consists of a *GetChallenge* followed by a *CompleteBAC* and an *EAC* constitutes a *CA* followed by a *TA*. The files 257 and 258 should be readable after a *BAC* (and *EAC*). File 257 contains machine readable zone (MRZ) data, i.e. name, date of birth, nationality, document number, etc. whereas file 258 contains a facial image. File 259 comprises biometric data like fingerprints or an iris scan, which are only readable after a *BAC* followed by an *EAC*. All other files should not be readable at any point in time.

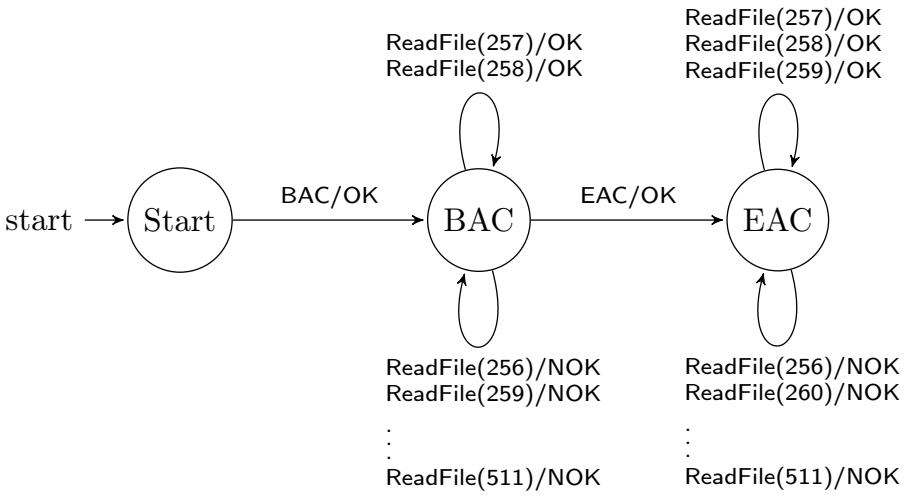


Figure 4.2: *Simplified model of the biometric passport*

4.3 Experiments

We have implemented and applied our approach to infer a model of the biometric passport described in Section 4.2. In this section, we first describe our experimental setup, thereafter its application and a validation of our technique.

We used an authentic biometric passport as *SUT*. The data on the chip could be accessed via a smart card reader; JMRTD[1] provided the API. We connected the *SUT* to an abstraction mapping, which performed a translation as described in Section 4.3.1. As before, the LearnLib library [131, 114] served as *learner*. For equivalence approximation we used the W-Method by Chow [41] implemented in LearnLib.

4.3.1 Abstraction Mapping

As described in Section 4.2, only the *ReadFile* message has a parameter called *file*, which can take on integer values in the range from 256 up to (and including) 511. Actually, each of these numbers has to be considered separately in the inference process, which would require a lot of time and memory space. By taking a closer look at the informal specification of the passport, we discovered that different files should be treated in the same way by the *SUT*. As one can see in Figure 4.2, files 257 and 258 should be readable after a *BAC*, 259 after a *BAC* followed by an *EAC* and the rest of the files should never be readable. Using this *a priori* knowledge about the passport, we can divide the values into three disjoint equivalence classes, which are:

- *ValidAfterBAC* refers to the files that can be read after a *BAC*, i.e. 257 and 258.
- *ValidAfterEAC* refers to the files that only can be read after a *BAC* followed by an *EAC*, i.e. 259.
- *NotValid* refers to the files that can never be read, i.e. all files except for 257, 258 and 259.

For the construction of the abstraction mapping, we apply the same technique as presented in Section 3.3. Initially, all values are in one large equivalence class. The mapper component translates an abstract value to a concrete one by randomly choosing an element within the equivalence class. If the numbers are partitioned incorrectly, then there are two values in the same class that will produce a different response, e.g. *BAC* followed by *ReadFile(257)* will result in an *OK* output while for *BAC* followed by *ReadFile(402)* a *NOK* is produced. This nondeterministic behavior will be detected by LearnLib, which will give an error message. As a result, we refine the partitioning and start from scratch.

4.3.2 Results

The inference performed by LearnLib needed about one thousand output queries (sequences of inputs) and one equivalence query, and resulted in a model $\mathcal{H}$

[1] http://jmrtd.org/

with five states and 55 transitions. Without our abstraction mapping, the Mealy machine would have had 1320 transitions, but also five states. The total learning time took less than one hour[2]. This is significantly shorter than deriving the model manually from the informal specs, which took about 5 hours. All results are summarized in Table 4.1. For presentation purposes, we have depicted the model as follows: (1) we removed self-transitions with *NOK* as output. Because the model is input-enabled all missing entries refer to this kind of transition. (2) Transitions with same source location, output symbol and next location (but with different input symbols) are merged by concatenating the input symbols, separated by a bar (|). The resulting transition diagram has five locations and 19 transitions as shown in Figure 4.3.

Output queries (sequences of inputs)	1078
Total input symbols used in output queries	4158
Average output query length	3.857
Equivalence queries	1
Total learning time	$<$ 60 minutes

Table 4.1: *Learning statistics*

The implementation of the biometric passport does not respond to a *Reset* input. For all other outputs the reaction time is dependent on the input symbol. If the waiting time for an output is too short, then an output symbol may be returned after a *timeout* has been assumed. In contrast, if the waiting time is too long, then the passport application crashes after certain inputs. As a solution, we changed the API of the *SUT*, so that it returns an *OK* symbol for each *Reset* input. By always returning an output symbol, we do not have to struggle with appropriate waiting times per input symbol. Instead, we wait until an output is received.

According to the passport specification, the implementation should be deterministic. However, surprisingly, the passport application sometimes exhibits nondeterministic behavior. LearnLib is restricted to infer behavior deterministic Mealy machines and cannot cope with nondeterministic behavior. Analyzing the external behavior of the system revealed that after a *GetChallenge*, *CompleteBAC*, *CA*, *TA* input sequence mostly an *OK* is returned, but in some rare cases it can also be a *NOK*. Together with Mostowski et al. we tried to examine the internal behavior of the application to understand where the nondeterminism originates from. During their work this problem has also been encountered, but it has never been reported. Because a *TA* call includes numerous complex and long calculations, a problem can arise at several places. Moreover, external circumstances may influence the produced results like connection to or temperature of the smart card reader. In the end, we could not clearly determine the fault location and had to accept that the inference can fail once in a while.

[2]The experiments have been carried out on a PC with an Intel Pentium M 1.86GHz processor and 1GB of RAM.

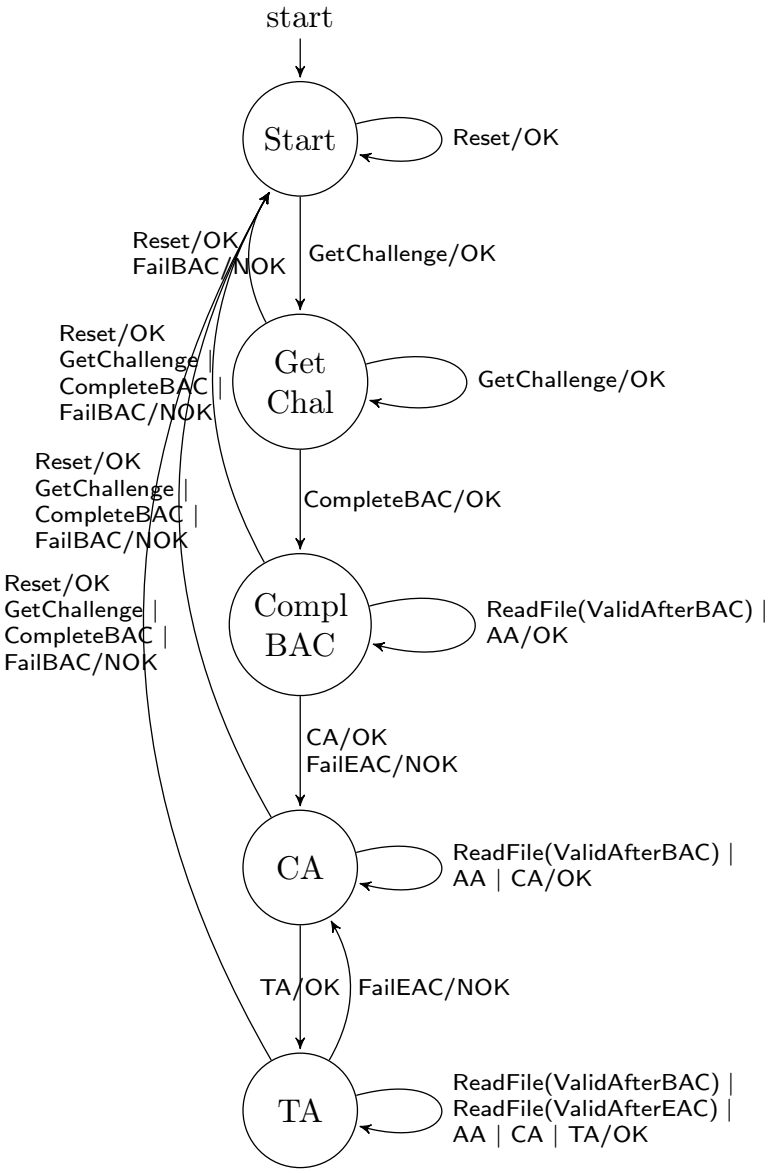


Figure 4.3: *Learned model $\mathcal{H}$ of the biometric passport*

4.3.3 The Behavior of the SUT

We assume that the behavior of the digital passport can be modeled in terms of a behavior deterministic Mealy machine $\mathcal{M}$. Clearly, due to the abstraction that we applied, the learned model $\mathcal{H}$ is not equivalent to $\mathcal{M}$: even the alphabets are different. Let $\alpha_{\mathcal{A}}(\mathcal{M})$ be the Mealy machine obtained from $\mathcal{M}$ via mapper $\mathcal{A}$ by renaming each action *ReadFile*(*file*) in accordance with the abstraction mapping defined in Section 4.3.1. We assume that also $\alpha_{\mathcal{A}}(\mathcal{M})$ is behavior deterministic. Since the SUT and the mapper together behave like $\alpha_{\mathcal{A}}(\mathcal{M})$, the learned model $\mathcal{H}$ should include the behavior of $\alpha_{\mathcal{A}}(\mathcal{M})$, i.e. $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. LearnLib constructs a deterministic automaton $\mathcal{H}$ and implements several algorithms that can be used to "approximate" inclusion queries, that is, to establish that the hypothesized machine $\mathcal{H}$ includes the behavior of the model $\alpha_{\mathcal{A}}(\mathcal{M})$ of the *teacher*. We have used the well-known W-method of [41] (see also [100]). This method assumes a

known upper bound on the number of states n of $\alpha_{\mathcal{A}}(\mathcal{M})$. Depending on n the W-method provides a test sequence of input symbols u with the property that the output produced by $\alpha_{\mathcal{A}}(\mathcal{M})$ in response to u is also included in the observations of $\mathcal{H}$. But assuming that we have established behavior inclusion of $\alpha_{\mathcal{A}}(\mathcal{M})$, i.e. $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, what have we learned about $\mathcal{M}$?

We reverse the abstraction mapping and construct a "concrete" model $\gamma_{\mathcal{A}}(\mathcal{H})$ of the passport as follows. We replace each *ValidAfterBAC* transition in $\mathcal{H}^{\mathcal{A}}$ by two transitions with the same source and target but with labels *ReadFile(257)* and *ReadFile(258)*, respectively. Similarly, we replace each transition with label *ValidAfterEAC* by an identical transition with label *ReadFile(259)*. Finally, we replace each transition with label *NotValid* by 253 identical transitions with labels *ReadFile(256)*, *ReadFile(260)*, *ReadFile(261)*, ..., *ReadFile(511)*, respectively.

Theorem 3.1 states that if $\alpha_{\mathcal{A}}(\mathcal{M})$ implements $\mathcal{H}$, then $\mathcal{M}$ implements $\gamma_{\mathcal{A}}(\mathcal{H})$. Provided that $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic, $\mathcal{M}$ is observation equivalent to $\gamma_{\mathcal{A}}(\mathcal{H})$, i.e. $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$, according to Lemma 2.1. This is indeed the case since for the biometric passport mapper $\mathcal{A}$ is output-predicting as it acts as the identity on outputs, $\mathcal{H}$ is deterministic and, accordingly, Lemma 3.4 states that $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.

4.3.4 Validation

To validate the learned model of the biometric passport, we compared it to a reference model taken from Mostowski et al. [118]. The specification is a LTS made in Haskell[3] and has to be transformed to a different format to allow comparison with the inferred Mealy machine described in the *DOT*[4] language.

For the comparison, we used JTorX [25], a tool to test whether the **ioco** testing relation holds between a given specification and a given implementation. Intuitively, an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is input-output conforming to specification $s \in \mathcal{LTS}(L_I, L_U)$ if any experiment derived from s and executed on i leads to an output from i that is foreseen by s. For a formal definition, we refer to [153]. We have supplied JTorX with the specification and implementation as LTS - represented in Aldebaran[5] format. The learned Mealy machine has been transformed to a LTS by splitting each transition into two with the input symbol on the first transition and the output on the second one connected by an additional state. As a result, the input-enabledness of the Mealy machine gets lost. To convert the learned LTS to an IOTS, JTorX adds self-loop transitions to the according states. Furthermore, we removed the output *OK* for a *Reset* input, because it is unknown by the specification, see Section 4.3.2.

According to JTorX, the implementation is **ioco** conforming to the specification, but not vice versa. This is not surprising as the learned model is input-enabled while the specification is not. For example, the specification does not specify a *CompleteBAC* input in the initial state while the learned implementation does, see

[3] http://www.haskell.org/
[4] http://www.graphviz.org/
[5] http://www.inrialpes.fr/vasy/cadp/man/aldebaran.html

Figure 4.4 for a fragment of the specification. We only show the inputs in the initial state with according outputs, because the entire model contains too many states and transitions. As one can see, the automaton corresponds to a Mealy machine. Except for the *Reset* input, each input is followed by an output. If we would transform the specification to a Mealy machine, it would not be input-enabled. Because LearnLib infers an input-enabled Mealy machine of the implementation, it contains more behavior than described by the specification, which is allowed by the **ioco** testing relation.

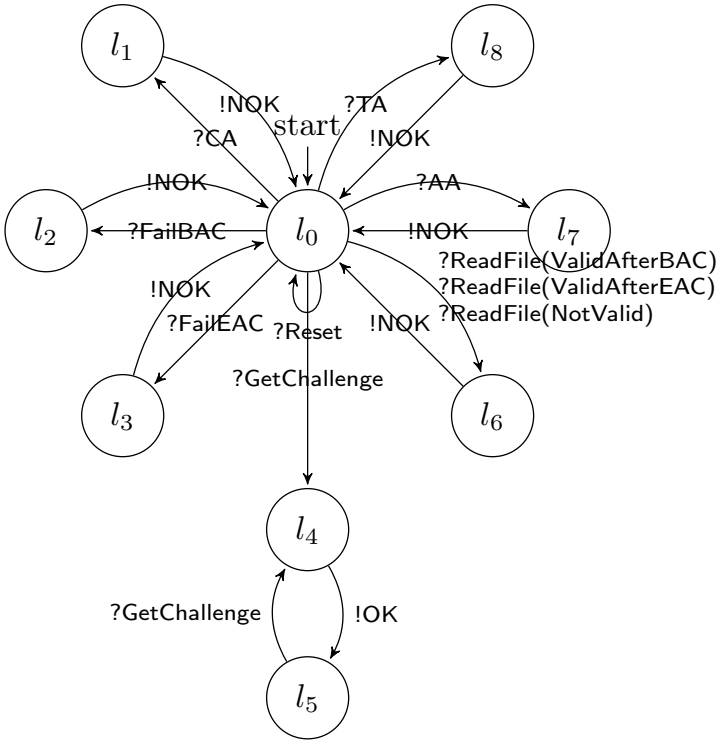


Figure 4.4: *Specification fragment of the biometric passport*

4.4 Conclusions and Future Work

Using regular inference and abstraction, we have managed to infer a model of the biometric passport that describes how the passport responds to certain input sequences. As mentioned in the introduction of this thesis, quite a number of papers have been written on regular inference of state machines. However, the number of real applications to reactive systems is still limited. The case study that we describe here is a small but real application. The new biometric passport is used by millions of people, and it is vital that the confidential information stored on this passport is well-protected. Our model, which slightly refines the earlier model of [118], may serve as a reference model for testing different implementations of the biometric passport.

The data abstraction that we applied when learning the passport may seem rather obvious (and indeed is much simpler than the abstractions applied in Chapter 3 of this thesis), but is nevertheless crucial for the successful application of our

learning framework. In order to prevent brute force attacks, the biometric passport only allows for about one input message per second. Without abstraction, the time needed to apply the framework (and in particular the approximation of equivalence queries via e.g. the W-method) would become prohibitively large. We have proven that under some reasonable assumptions about the behavior of the biometric passport, our abstraction does not lead to any loss of information.

The earlier model of [118] has been created manually in about 5 hours, whereas our model has been produced automatically in less than one hour. Our ambition is to further develop the learning framework, so that also for other applications it becomes feasible to mechanize and speed-up the time-consuming and error prone process of constructing reference models.

Due to the problems with the occasional nondeterministic behavior of the passport, an obvious topic for future research is to extend our approach to inference of nondeterministic systems. Such an extension will be essential, when doing more real-world case studies like this one.

If inferring an input-enabled Mealy machine is too time-consuming and we are only interested in parts of the implementation, we may extend our abstraction mappings with an *interface automaton (IA)* as suggested by [11]. An interface automaton [56] is a labeled transition system with input and outputs, where certain input actions may be illegal in certain states. When an input symbol or sequence generated by the learning algorithm is not allowed by the specified IA, this part of the implementation will not be inferred. By adding restrictions, we can focus on those parts of the implementation that are described by the specification.

Chapter 5

Formal Models of Bank Cards For Free

Software for banking or credit cards is developed using a very strict and regimented software engineering process. After all, this software is highly security-critical and patching is usually not an option. The software will be subjected to rigorous compliance tests and security certifications, possibly even costly Common Criteria certifications.

Establishing security here is often more difficult than just establishing correctness, or compliance with a standard. In checking compliance (e.g. for interoperability) the emphasis tends to be on the *presence of required functionality*: if some functionality is missing, the implementation is incorrect and it will not work correctly in all circumstances. Security on the other hand is also concerned with the *absence of unwanted functionality*; if an implementation provides more functionality than what is required, then it may be considered compliant – after all, it does what it is supposed to do – but it might be insecure, as it does *more* than what it is supposed to do, and this additional functionality may be a source of insecurity. This makes it hard to test for security bugs, and to discover them in the field: unlike functional bugs, security bugs may never show up under normal circumstances.

Testing of security applications using model-based testing techniques appears to be an interesting approach to test for security vulnerabilities [63], as a generalization of fuzzing. It does however require formal models that specify the intended behavior of the system. In this chapter, we show how standard active learning methods together with the abstraction techniques of Chapter 3 can be used to learn formal models of bank cards quickly without much effort. The inferred models can be a useful addition to testing or security evaluations of these products. The technique is not just applicable to bank cards, but can be applied to any smartcard.

5.1 Background: Smartcards and EMV

5.1.1 Smartcards

All smartcards follow the ISO/IEC 7816 standard [90]. Here communication is in master-slave mode: the terminal sends a command to the card, and the card returns a response, after which the terminal can send another command, etc. Commands and responses are simply sequences of bytes with a fixed format and meaning, called APDUs (application protocol data units).

The second byte in a command APDU is the *instruction byte*, and specifies the instruction that the smartcard is requested to perform. The last two bytes of a response APDU are the *status word*, which indicates if execution of the command went OK or if some error occurred. The ISO/IEC 7816 standard defines some standard instruction bytes and error codes.

Standard instructions we used to infer the behavior of bank cards include:

- the `SELECT` instruction to select which of the possibly several applications on the smartcard the terminal wants to interact with;
- the `VERIFY` instruction to provide a PIN code to the card for authentication of the cardholder;
- the `READ RECORD` instruction to read some data from the simple file system that the card provides;
- the `GET DATA` instruction to retrieve a specific data element from the card (for example the PIN try counter, which records how often the PIN can still be guessed);
- the `INTERNAL AUTHENTICATE` instruction to authenticate the card; the terminal supplies a random number as argument to this command which the smartcard then encrypts or signs to prove knowledge of a secret key.

For the purposes of this research the difference between the 'files' retrieved using `READ RECORD` and the 'data elements' retrieved using `GET DATA` is not important.

5.1.2 EMV

Most smartcards issued by banks or credit cards companies adhere to the EMV (Europay-MasterCard-Visa) standard [61]. This standard is defined on top of ISO/IEC 7816. It uses some of standard instruction bytes (incl. those listed above), but also defines additional ones specific to EMV, including:

- the `GENERATE AC` instruction to let the card generate a so-called application cryptogram (AC);
- the `GET PROCESSING OPTIONS` instruction to initialize the transaction, provide the necessary information to the card and retrieve the capabilities of the card.

A normal EMV session consists of the following steps:

1. *selection of the desired application on the smartcard using* `SELECT`. There may be several applications on a smartcard. Some bank cards will provide multiple EMV-applications for different uses, e.g. one to be used by ATMs and one to be used by a hand-held reader for internet banking.

2. *initialization of the transaction using* `GET PROCESSING OPTIONS`. The terminal provides the card with data, specified in the response to the selection of the application. The card initializes the transaction and sends a response containing its capabilities.

3. *optionally: cardholder verification and/or card authentication.* Card authentication can, for example, be done using a challenge-response mechanism (called DDA in the EMV standard) by invoking the `INTERNAL AUTHENTICATE` instruction. Cardholder verification can be done offline by checking the PIN code using the `VERIFY` instruction; here the PIN can be sent to the smartcard either in plaintext or encrypted. Checking the PIN can also be done online, in which case the PIN is sent to the bank back-end to check it.

4. *the transaction.* For the actual transaction one or two cryptograms are requested using the `GENERATE AC` instruction, as discussed in more detail below.

5. *scripting.* After completing a transaction, the terminal may send additional Issuer-to-Card scripting commands, that allow the issuer to update cards in the field.

The cryptograms generated for a transaction can have one of the following three types:

- an authorization request cryptogram (ARQC), which is a request to perform a transaction online;
- a transaction certificate (TC), which indicates acceptance of a transaction;
- an application authentication cryptogram (AAC), which indicates rejection of a transaction.

All these cryptograms contain a MAC (message authentication code), a hash over some data encrypted with a secret key.

An EMV transaction involves at most two of these cryptograms. The types of these cryptograms depend on the transaction. EMV transactions can be offline or online. For an offline transaction, the terminal sends data about the transaction to the card, and the card returns a TC to approve the transaction. For an online transaction, the card first provides an ARQC that is sent back to the issuing bank. The bank's response is sent to the card, which will then return a TC if the response is correct. Every transaction by the card is identified by a unique value of the application transaction counter (ATC), that the card keeps track of.

The terminal requests these cryptograms using the `GENERATE AC` command. The terminal will indicate which type of cryptogram it wants, but the card may

return a different type than requested. For example, when the terminal requests a TC, the card may return an ARQC (namely in case the card wants the terminal to go online to get approval for the transaction from the bank) or it may decline the transaction by responding with an AAC.

The EMV protocol as described above is also used for internet banking in the EMV-CAP protocol. Here bank customers use a handheld smartcard reader with a small display in which they insert their bank card. EMV-CAP is a proprietary standard of MasterCard. Unlike the EMV specs, the EMV-CAP specs are not public, but they have been largely reverse-engineered [60, 150]. In EMV-CAP the card is requested for an ARQC to authorize an transaction (e.g. an internet bank transfer). This is then followed by a request for an AAC, thus completing (or, more precisely, aborting) a regular EMV transaction so that the card is left in a 'clean' state.

5.2 Setup and Procedure

We used authentic bank cards as *SUT*/*teacher*. Access to the smartcards was realized via a standard smartcard reader and a testing harness discussed in Section 5.2.1. We connected the *SUT* to the LearnLib library [131, 114], which served as *learner*, see Figure 5.1. In our experiments we used the LearnLib random test suite with 1000 test traces of length 10 to 50 as equivalence oracle. We verified our results with the W-Method by Chow [41] by checking if it will find at least one more state than the random test suite.

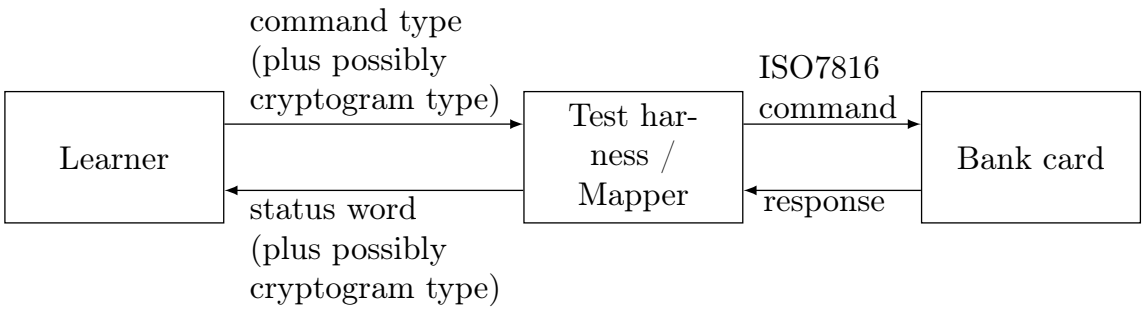


Figure 5.1: *Set-up*

For our tests we used a collection of MasterCard and Visa branded debit and credit cards from the Netherlands, Germany, Sweden and the UK. All the MasterCard credit cards contain a MasterCard application, whereas on the bank cards there is a Maestro application. Both these applications are used for payments in shops and to withdraw cash from ATMs. The Dutch bank cards also contain a SecureCode Aut application, which is used for online banking with a handheld EMV-CAP reader provided by the bank. The Visa branded debit card contains the Visa Debit application.

5.2.1 Test Harness

As illustrated in Figure 5.1, our test harness[1] translates the abstract command (from the input alphabet of our Mealy machine model) to a concrete command APDU, and translates a response APDU to a more abstract response (in the output alphabet of the Mealy machine model). It supports the commands listed in Section 5.1.2.

The test harness is just over 300 lines of Java code. Most of this code is generic code to set-up a connection to the smartcard reader. A regular smartcard reader was used, and communication was performed using the standard Java Smart Card I/O library. The code specific to EMV is just over 100 lines of code, and consists of 15 methods that define some command APDU to be sent to the card. The input alphabet corresponds to these 15 methods.

For many parameters of these commands the test harness uses some fixed value, for instance for the random number sent as argument of `INTERNAL AUTHENTICATE`, the payload data for the cryptograms generated by the card, and the (correct) guess for the PIN code. One would not expect a different random number to affect the control flow of the application in any meaningful way, so by fixing values here we are unlikely to miss interesting behavior. Note that we have two different payloads when requesting the cryptograms due to the difference between the first and second request for a cryptogram. As these payloads are different, both a correct and an incorrect payload is used when requesting cryptograms. Obviously, entering an incorrect PIN code would affect the control flow, but learning about the behavior in response to incorrect PIN guesses is very destructive as it will quickly block the card.

For several commands different variants are provided by the test harness:

- For the commands `GET DATA`, `READ RECORD` and `GET PROCESSING OPTIONS`, both a variant with correct arguments and one with incorrect arguments is provided. E.g., for `GET DATA` we have variants requesting a data element that is present or one that is not.

- For the `GENERATE AC` command 6 variants are provided, as there are 3 cryptogram types, each of which can be used with one of 2 sequences of arguments (one for the first and one for the second cryptogram).

The test harness does not output the entire response of the smartcard to the *learner*. It only returns the 2 byte status word, but not any additional data returned by the card. For most commands, like `GET PROCESSING OPTIONS`, this additional data returned will always be the same, so there is not much interest in learning it. The only exception to this is the `GENERATE AC` command: here the test harness does return the type of cryptogram that was returned by the card (but not the cryptogram itself; as this is computed using a cryptographic function on the input and the card's ATC, the response will never be the same and there is nothing we could hope to learn from it).

A limitation of our test harness is that we do not know the bank's secret cryptographic keys that are needed to complete one 'correct' path of the protocol,

[1] Available from `http://www.cs.ru.nl/~joeri/`

namely the path where the card produces an ARQC as first cryptogram and a TC as second. For this a correct reply to the first ARQC is needed, which requires knowledge of the cryptographic keys used by the bank's back end.

To be able to include the `VERIFY` command in the learning, the PIN code of the corresponding card has to be known. We did not try to learn the behavior of the card in response to incorrect PIN codes, to avoid blocking the card. The cards we used are real bank cards for which we cannot reset the PIN. (With access to functionality to reset the PIN, which the issuing bank might have, one could also try to learn the behavior in response to incorrect PINs.) The German card only supported encrypted PIN verification. Since the public key of MasterCard is needed for this, we were unfortunately not able to use `VERIFY` with this card.

The Visa branded card can perform the `GET DATA` command to retrieve the current value of the ATC. This functionality is used by a *mapper* component in the sense of Chapter 3 to be able to learn the transitions where a counter is increased. The *mapper* is integrated in the test harness of the *SUT*, and uses variable *lastATCReceived* to keep track of the value of the counter. The `GET DATA` command only returns the current value of the ATC if the Visa Debit application is selected. Since the mapper depends on the value of the ATC, the Visa Debit application is automatically selected by the test harness when a reset is performed. We assume that the initial state of the mapper coincides with the ATC value of the SUT. We can implement this by performing the `GET DATA` command right after the reset. The mapper retrieves the value of the ATC when it receives an output from the SUT and stores it in the *lastATCReceived* variable. We formally define the update function of the mapper by the following transition:

$$\begin{aligned} &\textbf{event } \mathit{Output}(\mathit{ATC}) \textbf{ when} && \mathsf{TRUE} \textbf{ do} \\ &\langle \mathit{lastATCReceived} \rangle := && \langle \mathit{ATC} \rangle \end{aligned}$$

For all input event terms no state variables are updated, and we have trivial transitions of the form **event** $\mathit{Input}()$ **when** TRUE **do** $\langle\rangle := \langle\rangle$.

The mapper adds the difference with the *lastATCReceived* variable to the abstract response, e.g. '1' in an output indicates the ATC was increased by one in this transition. We define the relation between concrete and abstract output symbols by the event abstraction $\mathit{Output}(e)$, where $\mathit{Output}(\mathit{ATC})$ can be any output event term, and

$$e = \mathit{ATC} - \mathit{lastATCReceived}.$$

Since the input event terms carry no parameters, the event abstraction for these terms is trivial.

5.2.2 Trimming the Inferred State Diagrams

The state diagrams returned by LearnLib as .dot file look quite unintelligible at first sight, because there are so many transitions: for each state, one for *every* possible command. However, many transitions from a given state are errors and simply return to the same or an error state (e.g. the 'Selected' or 'Finished' state). By simply collapsing all these transitions into one transition marked 'Other' or

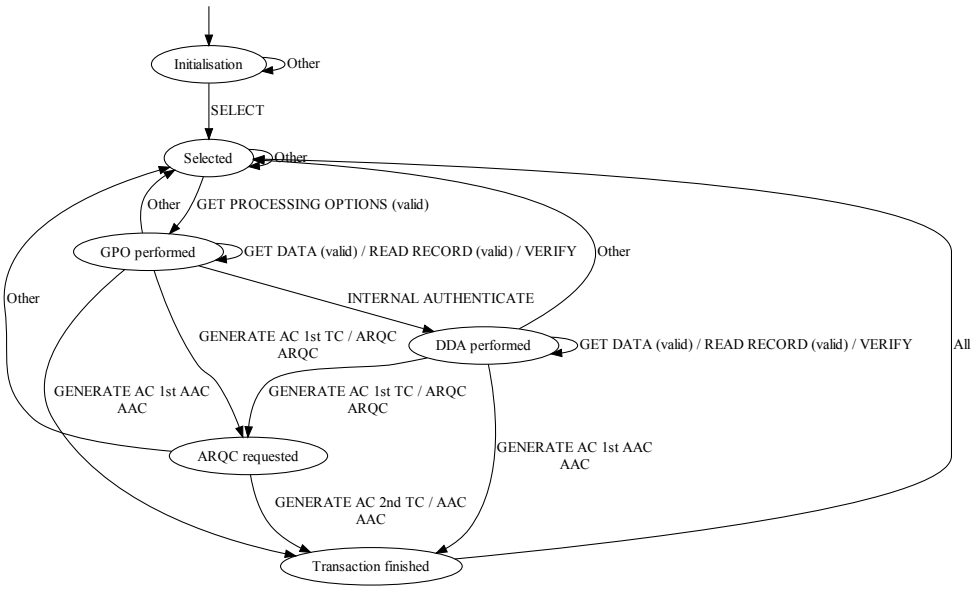


Figure 5.2: *Automaton of Dutch Maestro application. Just to highlight one observation that can be made from this diagram: the* `VERIFY` *operation, i.e. the verification of the PIN code by the smartcard, is optional; this makes sense because the terminal may check the PIN code with the bank (so-called online PIN verification), or choose not to verify the PIN at all.*

'All', and drawing multiple transitions between the same states with different labels as one transition with a set of labels, we obtain simple automata such as Figures 5.2, 5.3, 5.4 and 5.5. In these figures the responses are omitted for readability. We simply obtained these by manually editing the .dot files. This could easily be automated. At the same time we chose meaningful names for the different states.

The transition labels for `GENERATE AC` commands indicate (i) if it is the 1st or 2nd request for a cryptogram in this session (i.e. whether the argument is for the first or second request), (ii) the type of cryptogram that was requested (ARQC, AAC, or TC), and (iii) the type of cryptogram that was returned. E.g. `GENERATE AC 1st ARQC ARQC` means the type requested was ARQC, the arguments supplied for the first request, and the type returned was an ARQC. We have combined arrows if different parameters yield the same response; e.g. `GENERATE AC 2nd TC/AAC AAC` means that requests for a TC or AAC, with the arguments for the second request, both result in an AAC.

5.3 Results

We learned models of EMV applications on bank cards issued by several Dutch banks (ABN-AMRO, ING, Rabobank) and one German bank (Volksbank), and on MasterCard credit cards issued by Dutch and Swedish banks (SEB, ABN-AMRO, ING) and of one UK Visa Debit card (Barclays). The Dutch bank cards contain

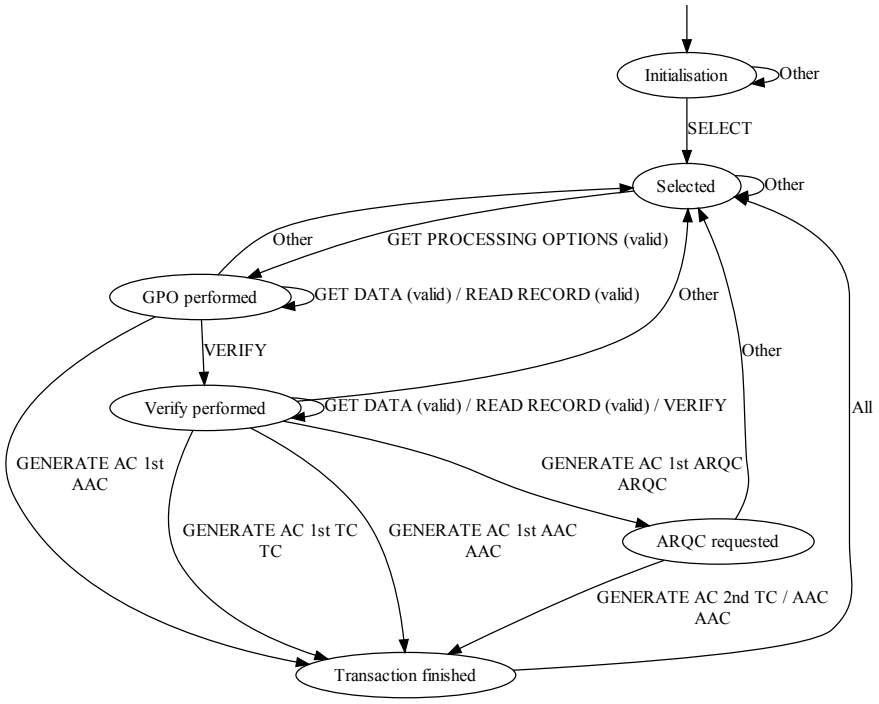


Figure 5.3: *Automaton of Dutch SecureCode Aut application. Note that here the* `VERIFY` *operation – i.e. verification of the PIN code – must be passed successfully before cryptograms can be generated, except for the AAC cryptogram to abort the session.*

two EMV applications, one for internet banking (SecureCode Aut) and one for ATMs and Point-of-Sales (Maestro). All cards resulted in different models, with as only exception that the Maestro applications on all Dutch bank cards were identical, as were the SecureCode Aut applications. An educated guess would be that these implementations come from the same vendor.

To learn the models LearnLib performed between 855 and 1695 output queries (sequences of inputs) for each card and produced models with four to eight states. The total learning time depended on the algorithm and corresponding parameters used for equivalence approximation. The time needed to construct the final hypothesis was less than 20 minutes for every card.

When analyzing the state diagrams for the different categories, we made the following observations.

The state diagrams for the ABN-AMRO and ING credit cards are very similar. There are only a few subtle differences, e.g in the initial state different error codes are returned in response to some instructions. Also the handling of the `INTERNAL AUTHENTICATE` instruction differs: both cards respond with the error `6D00` ('Instruction code not supported or invalid'), indicating that the instruction is not supported, but for the ING card this does not have any influence on the state, whereas the ABN-AMRO card is 'reset' to the 'Selected' state.

Comparing the Maestro (Figure 5.2) and the SecureCode Aut application (Figure 5.3) on the Dutch bank cards, we can observe the following:

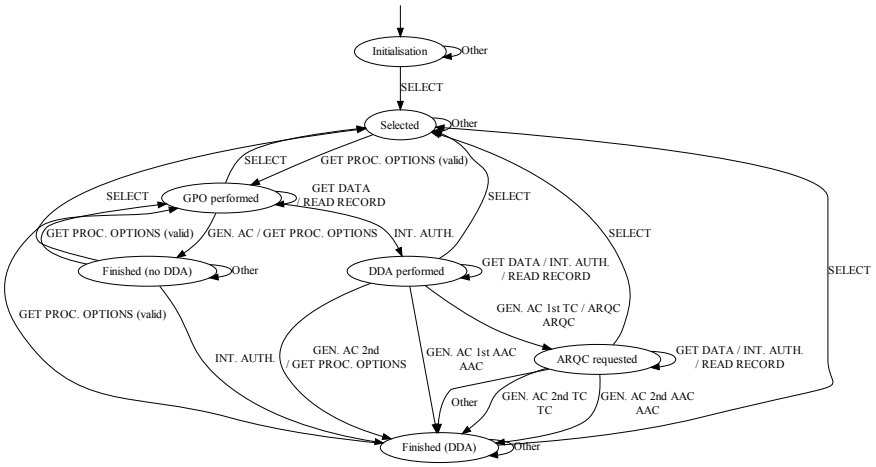


Figure 5.4: *Automaton of Maestro application on Volksbank bank card*

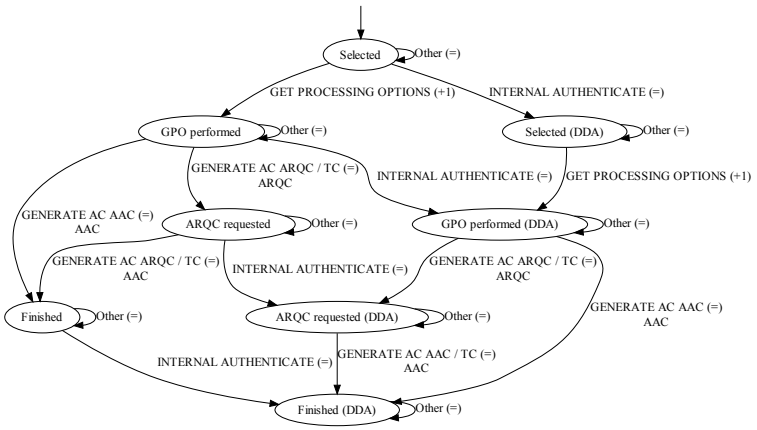


Figure 5.5: *Automaton of Visa Debit application on Barclays card. Note that the* `INTERNAL AUTHENTICATE` *can be performed at any stage of the protocol.*

1. In both applications, if data that is not available is requested, either using the `READ RECORD` or the `GET DATA` instruction, the application returns to the 'Selected' state. This seems a bit strict, as the terminal has no way of knowing whether certain data that can be retrieved using `GET DATA` is available. Apparently, here the developers have chosen a 'safe by default' approach. Though this seems a sensible approach, one can imagine this can lead to compatibility problems with terminals that expect certain data to be present on the card while it is not, as the card will reset to a state that the terminal might not expect.

2. With the SecureCode Aut application it is possible, after successfully verifying the PIN code, to request a TC cryptogram using the `GENERATE AC` instruction. This is surprising, as this does not have any meaning in EMV-CAP: in an EMV-CAP session the terminal must always first ask for an

ARQC (as explained at the end of Section 5.1). One would expect that requesting a TC cryptogram type would result in an error (as e.g. happens when a second ARQC is requested) or in an AAC being returned to abort the session (as e.g. happens when any type of cryptogram is requested before PIN verification). Still, it does not seem that this spurious TC cryptogram can be exploited to cause a security vulnerability, at least insofar as we know the EMV-CAP protocol [60, 150].

3. The error code that is given in response to the `INTERNAL AUTHENTICATE` instruction is different depending on the state in the SecureCode Aut application. In those states where it is possible in the Maestro application to perform this action, the error code is `6987` ('Expected secure messaging data objects missing'), while in the other states, an error code `6985` ('Usage conditions not satisfied') is returned.

Compared to the cards considered before, the Volksbank card handles things a bit differently (see Figure 5.4):

1. Where the other cards return to the 'Selected' state when an error occurs, the Volksbank card goes into a 'Finished' state. From a 'Finished' state there is one transition using the `SELECT` command to get to the 'Selected' again, and one to get to the 'GPO performed' state using a valid `GET PROCESSING OPTIONS` command.

2. Data authentication using DDA is also handled differently with this card. First, the card forces DDA to be performed, i.e. if no `INTERNAL AUTHENTICATE` command is given, transactions cannot be performed: the `GENERATE AC` command will then always return an error. Also, it is possible to perform DDA even if the card is in a 'Finished' state. This suggests that the `INTERNAL AUTHENTICATE` command is handled separately from the other commands and keeps it's own state to indicate whether it is already performed. Below we compare this with what the MasterCard's specifications say.

3. If in the first `GENERATE AC` a TC is requested, the card indicates it wants to go online by returning an ARQC. However, after an ARQC is returned the first time, when requesting a TC in the second `GENERATE AC`, this is actually returned. This seems odd since one would expect this request to fail (i.e. an AAC to be returned), as we did not provide a valid response from the bank.

5.3.1 Difference with MasterCard's Specifications

The Maestro and MasterCard-branded applications should all conform to MasterCard's Paypass-M/Chip specification[2]. This specification does specify a state

[2]This specification is for dual interface (contact and contactless) cards, rather than contact-only cards, but states that the state diagram for contact-only cards is the same, except that it has one transition less [88, p. 98].

diagram, which has only 5 states, whereas the models we obtained for Maestro cards have 6 or 7 states.

In the state diagram specified by MasterCard the operation `INTERNAL AUTHENTICATE` has no effect on the state, meaning that this operation – i.e. performing DDA – is optional and can be done *any number of times*. In contrast, the model learned for the Dutch Maestro card says that this operation can be done *at most once* before cryptograms can be generated, and the model for the Volksbank Maestro card says that it must be done *exactly once* before cryptograms can be generated.

Another difference between the state diagram of the Volksbank card and the one specified by MasterCard is the presence of the 'Finished (no DDA)' state, which seems to be a spurious dead-end in the behavior of the Volksbank card, as it does not lead to a normal protocol run which ends where one or two cryptograms are generated.

As these cards carry the Maestro or MasterCard logo, they must have undergone some certification. Assuming that their certification has not missed potential compatibility problems caused by these deviations from MasterCard's specification, this does suggest that this process does not include checking for implementation of the exact state machine.

5.3.2 Different Choices in the Visa Branded Card

In the models of the MasterCard applications there exists an 'Initialization' state from which the applications can be selected on the smartcard. Since with the Visa branded card the test harness automatically selects the Visa Debit application, this initialization state is not included in the learned models and the initial state is 'Selected'.

The Visa branded card is quite different from the others. For example, with the Visa card the commands `GET DATA`, `READ RECORD` and `VERIFY` are allowed in all states, even before the transaction is initialized with `GET PROCESSING OPTIONS` and after the actual transaction is started with a `GENERATE AC` command or even finished. These commands are thus apparently completely independent from the state of the card. Also, DDA can be performed, by an `INTERNAL AUTHENTICATE`, completely independent of any other actions, again even during and after a transaction.

In the model it can be seen from the additional information added by the mapper that only two transitions increase the ATC. This indicates that the ATC is increased when performing a successful `GET PROCESSING OPTIONS` command (i.e. `9000` is returned as the status word).

5.4 Related Work

Protocol fuzzing is an increasingly popular technique to test for security vulnerabilities. Simpler forms of protocol fuzzing consider only the format of messages, and then fuzz the different fields, often simply to try and crash an implementa-

tion. More advanced fuzzers, such as Snooze [22] and Peach[3], take a state-based approach and also use a state machine describing the protocol as basis for fuzzing. Models such as we obtain could be the basis for more thorough state-based fuzzing by such tools. Model-based testing has already been applied to security, including for smartcards, for instance using UMLSec models [94]. For EMV smartcards, there have been successful experiments with protocol fuzzing based on state models at a commercial test lab [99]; here models were constructed by hand.

There is a growing interest in model inference, or more generally automated protocol reverse engineering, for security testing and analysis; see [102] for a survey and a proposed classification of approaches, and [78] for a discussion of future directions in combining model inference and model-based testing for security. In automated techniques for protocol reverse engineering one can distinguish approaches that try to infer either message formats (e.g. [51]) or protocol state machines (as we do, and [84]), or both (e.g. [49]). Another classification is that some approaches use 'passive learning', where the *learner* just observes traffic between other parties (e.g. [51, 84, 49]), and 'active learning', i.e. where the *learner* actively takes part in the traffic in order to learn (as we do). A fundamental limitation of passive learning is that the quality of the model depends on the traffic that is observed. It will typically not provide good insight into the possibility of unwanted behavior. It is therefore natural to follow such passive learning by protocol fuzzing to actively look for any such behavior. Indeed, fuzzing based on the inferred model is considered as final stage in [84, 49].

In the previous chapter we have shown that active learning was successfully used to infer models for the electronic passport, confirming that the right 'files' are accessible at different stages of the protocol. The models inferred in this chapter are more complex than the one of the passport, and the models reveal interesting differences between various cards. Additionally, we could learn which command increased the ATC using a mapper component.

5.5 Conclusions and Future Work

We have demonstrated that after defining a simple test harness/mapper component, we can easily obtain useful state machine models for banking smartcards using learning and simple abstraction techniques as presented in Chapter 3 and in [131]. After some trimming, the models obtained are easy to understand for anyone familiar with the EMV standard, and clearly highlight some of the central decisions taken in an implementation.

Differences in the models obtained for different cards may be inconsequential differences that exploit the implementation freedom allowed by the underspecification in the EMV specs, but can really affect the security conditions imposed (for example, the difference between Figures 5.2 and 5.3 in requiring PIN code verification). To determine which is which, we have relied on ad-hoc manual work and human intelligence - the models obtained are easy to inspect visually. This step could even be automated if security conditions are expressed as temporal logic formulae.

[3] `http://peachfuzzer.com`

Differences in the state diagrams do not necessarily mean that implementations are not secure or that they cannot be regarded as compliant to the standard. The diagrams are a helpful aid in deciding whether this is the case. However, this decision then inevitably relies on an informal understanding of the standard and the essential security requirements. One would like to see more objective criteria for this, especially as security protocols are notoriously brittle and deciding what constitutes a secure refinement of the specification is not always easy.

The complexity of the standards involved make such models very valuable. In fact, finite state machine models such as we obtain would be a useful addition to the official specifications. Despite the length of the EMV specs [61] (of over 700 pages), state diagrams describing the smartcard are conspicuously absent. A state diagram is specified in MasterCard's specification [88], but most of the cards we analyzed actually did not conform to it in the sense that they were not bisimilar. The differences between e.g. Figures 5.2 and 5.4 show the considerable leeway there is between different implementations of the same spec. One would expect (and hope?) that engineers developing, testing, or certifying EMV smartcards do have such state diagrams, either in the official documentation or just scribbled on a whiteboard.

The models learnt did not reveal any security issues. Indeed, one would not expect to find any in smartcards such as we considered, which should have undergone rigorous security evaluations and tests. Still, we do notice some peculiarities (notably that the Volksbank card is still willing to return a TC even after failed issuer authentication). We believe that our approach would be useful as part of security evaluations, because it increases the rigor and confidence provided and it can save a lot of expensive and boring manual labor.

Here it helps that LearnLib learns the behavior blindly, in a completely haphazard way, without any of the preconceptions or expectations about what the 'normal' behavior is that a human tester or code reviewer might have. The tool learns about *all* the possible behavior. This is an advantage for security, as security bugs often occur under unusual conditions, when someone does something unexpected.

Still, the hand-coded test harness we developed does make some assumptions about the functionality that the card provides. The test harness implements the basic operations for EMV, and LearnLib then only learns all the possible behaviors given these operations. A deliberately introduced backdoor would thus not be detected, but we conjecture that any mistake in the implementation of the internal state and the associated control flow in the smartcard code would.

For future work, we want to try out our technique on more standard networking protocols such as SSH or TLS/SSL. This might be more fruitful in the sense that we can expect implementation bugs to be more common here, as these protocols are more complex and the code is less rigorously developed and tested than smartcard code. In the field of EMV, we plan to see if learning techniques can be used to assess EMV test suites provided by commercial testing companies; models learned from such test suites, using passive rather than active learning, could provide coverage criteria to assess their quality.

Part III

Active Learning of Scalarset Mealy Machines

Chapter 6
Automata Learning through Counterexample-Guided Abstraction Refinement

In this chapter, we present an algorithm that is able to compute appropriate abstractions for a restricted class of system models. We also report on a prototype implementation of our algorithm named Tomte, after the creature that shrank Nils Holgersson into a gnome and (after numerous adventures) changed him back to his normal size again. Using Tomte, we have succeeded to learn *fully automatically* models of several realistic software components, including the biometric passport and the SIP protocol.

Nondeterminism arises naturally when we apply abstraction: it may occur that the behavior of a *teacher* or SUT is fully deterministic but that due to the mapper (which, for instance, abstracts from the value of certain input parameters), the SUT appears to behave nondeterministically from the perspective of the *learner*. We use LearnLib as our basic learning tool and therefore the abstraction of the SUT may not exhibit any nondeterminism: if it does then LearnLib crashes and we have to refine the abstraction. This is exactly what has been done repeatedly during the manual construction of the abstraction mappings in the previous chapters. We formalize this procedure and describe the construction of the mapper in terms of a counterexample-guided abstraction refinement (CEGAR) procedure, similar to the approach developed by Clarke et al. [45] in the context of model checking.

Our algorithm applies to a class of extended finite state machines, which we call scalarset Mealy machines, in which one can test for equality of data parameters, but no operations on data are allowed. The notion of a scalarset data type originates from model checking, where it has been used for symmetry reduction [89]. In this chapter, we focus on learning SUTs that may only remember the first and last occurrence of a parameter. How to dispose of this restriction will be handled in the next chapter. We expect that our CEGAR based approach can be further extended to systems that may apply simple or known operations on data, using technology for automatic detection of likely invariants, such as Daikon [62].

The fact that we are able to learn models of systems with data fully automatically is a major step towards a practically useful technology for automatic

learning of models of real-world systems. The Tomte tool and all models that we used in our experiments are available via `http://www.tomte.cs.ru.nl/`. A full version of this article [3] including proofs is available via `http://www.italia.cs.ru.nl/publications/`.

6.1 The World of Tomte

Our general approach for using abstraction in automata learning is phrased most naturally at the semantic level. However, if we want to devise effective algorithms and implement them, we must restrict attention to a class of automata and mappers that can be finitely represented. In this section, we describe the class of SUTs that our Tomte tool can learn, as well as the classes of mappers that it uses.

6.1.1 Scalarset Mealy Machines

Below we define *scalarset Mealy machines*. The scalarset data type was introduced by Ip and Dill [89] as part of their work on symmetry reduction in verification. Operations on scalarsets are restricted so that states are guaranteed to have the same future behaviors, up to permutation of the elements of the scalarsets. On scalarsets no operations are allowed except for constants, and the only predicate symbol that may be used is equality.

We assume a universe $\mathcal{V}$ of *variables*. Each variable $v \in \mathcal{V}$ has a domain $\mathsf{type}(v) \subseteq \mathbb{N} \cup \{\bot\}$, where $\mathbb{N}$ is the set of natural numbers and $\bot$ denotes the undefined value. A *valuation* for a set $V \subseteq \mathcal{V}$ of variables is a function ξ that maps each variable in V to an element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for V. We also assume a finite set C of *constants* and a function $\gamma : C \to \mathbb{N}$ that assigns a value to each constant. If $c \in C$ is a constant then we define $\mathsf{type}(c) = \{\gamma(c)\}$. We require that, for all $c, c' \in C$, $\gamma(c) = \gamma(c')$ implies $c = c'$. A *term* over V is either a variable or a constant, that is, an element of $C \cup V$. We write $\mathcal{T}$ for the set of terms over $\mathcal{V}$. If t is a term over V and ξ is a valuation for V then we write $[\![t]\!]_\xi$ for the value to which t evaluates:

$$[\![t]\!]_\xi = \begin{cases} \xi(t) & \text{if } t \in V \\ \gamma(t) & \text{if } t \in C \end{cases}$$

A *formula* φ over V is a Boolean combination of expressions of the form $t = t'$, where t and t' are terms over V. We write $\mathcal{G}$ for the set of all formulas over $\mathcal{V}$. If ξ is a valuation for V and φ is a formula over V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. We assume a set E of *event primitives* and for each event primitive ε an arity $\mathsf{arity}(\varepsilon) \in \mathbb{N}$. An *event term* for $\varepsilon \in E$ is an expression $\varepsilon(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $n = \mathsf{arity}(\varepsilon)$. We write $\mathcal{ET}$ for the set of event terms.

Event signature An *event signature* Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are finite sets of event terms such that $T_I \cap T_O = \emptyset$ and each term in $T_I \cup T_O$ is of the form $\varepsilon(p_1, \ldots, p_n)$ with $p_1, \ldots, p_n$ pairwise different variables with $\mathsf{type}(p_i) \subseteq \mathbb{N}$, for each i. We require that the event primitives as well as the variables of different

event terms in $T_I \cup T_O$ are distinct. We refer to the variables occurring in an event signature as *parameters*.

Definition 6.1 (Scalarset Mealy machine) A *scalarset Mealy machine (SMM)* is a tuple $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$, where

- $\Sigma = \langle T_I, T_O \rangle$ is an event signature,
- $V \subseteq \mathcal{V}$ is a finite set of state variables, with $\bot \in \mathsf{type}(v)$, for each $v \in V$; we require that variables from V do not occur as parameters in Σ,
- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $\Gamma \subseteq L \times T_I \times \mathcal{G} \times (V \to \mathcal{T}) \times \mathcal{ET} \times L$ is a finite set of transitions. For each transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_l), l' \rangle \in \Gamma$, we refer to l as the *source*, g as the *guard*, ϱ as the *update*, and l' as the *target*. We require that g is a formula over $V \cup \{p_1, \ldots, p_k\}$, for each v, $\varrho(v) \in V \cup C \cup \{p_1, \ldots, p_k\}$ and $\mathsf{type}(\varrho(v)) \subseteq \mathsf{type}(v)$, and there exists an event term $\varepsilon_O(q_1, \ldots, q_l) \in T_O$ such that, for each i, u_i is a term over V with $\mathsf{type}(u_i) \subseteq \mathsf{type}(q_i) \cup \{\bot\}$,

We say $\mathcal{S}$ is *deterministic* if, for all distinct transitions $\tau_1 = \langle l_1, e_1^I, g_1, \varrho_1, e_1^0, l_1' \rangle$ and $\tau_2 = \langle l_2, e_2^I, g_2, \varrho_2, e_2^0, l_2' \rangle$ in Γ, $l_1 = l_2$ and $e_1^I = e_2^I$ implies $g_1 \wedge g_2 \equiv \mathsf{false}$.

Example 6.1 In the introduction of this thesis we have already seen an example of a scalarset Mealy machine, namely the login system in Figure 1.3. This extended finite state machine in Mealy machine format can be represented as a scalarset Mealy machine $\mathcal{S} = \langle \langle T_I, T_O \rangle, V, L, l_0, \Gamma \rangle$, where $T_I = \{\mathsf{Register}(id0, pw0), \mathsf{Login}(id1, pw1), \mathsf{Logout}\}$, $T_O = \{\mathsf{OK}, \mathsf{NOK}\}$, $V = \{\mathsf{ID}, \mathsf{PW}\}$, $L = \{\mathsf{INIT}, \mathsf{OUT}, \mathsf{IN}\}$, $l_0 = \mathsf{INIT}$, and the set of transitions Γ can easily be associated to the transitions in the diagram. □

A scalarset Mealy machine $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ can be viewed as a specific type of symbolic Mealy machine. Let *loc* be a fresh variable with type L. Then the symbolic Mealy machine associated to $\mathcal{S}$ is the tuple $\mathcal{SM}_{\mathcal{S}} = \langle \Sigma, V \cup \{loc\}, \Theta, \Delta \rangle$, where

$$\Theta \equiv loc = l_0 \wedge \bigwedge_{v \in V} v = \bot$$

and for each transition

$$\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_l), l' \rangle \in \Gamma$$

with corresponding event term $\varepsilon_O(q_1, \ldots, q_l) \in T_O$, Δ contains a transition

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \ldots, p_k)\ \textbf{\textit{when}}\ \varphi\ \textbf{\textit{event}}\ \varepsilon_O(q_1, \ldots, q_l),$$

where

$$\varphi \equiv loc = l \wedge g \wedge \bigwedge_{v \in V} v' = \varrho(v) \wedge \bigwedge_{i=1}^{l} q_i = u_i' \wedge loc' = l',$$

where, for $1 \leq \mathrm{i} \leq \mathrm{l}$, u'_i is the term obtained from u_i by replacing each variable $v \in V$ by v'.
With abuse of notation, we write $[\![\mathcal{S}]\!]$ for $[\![\mathcal{S}\mathcal{M}_{\mathcal{S}}]\!]$.

Our tool can infer models of SUTs that can be defined using deterministic SMMs that only record the first and the last occurrence of an input parameter.

Definition 6.2 (Restricted scalarset MMs) Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a SMM. Variable v *records the last occurrence* of input parameter p if for each transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, if $p \in \{p_1, \ldots, p_k\}$ then $\varrho(v) = p$ else $\varrho(v) = v$. Moreover, $\varrho(w) = v$ implies $w = v$. Variable v *records the first occurrence* of input parameter p if for each transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, if $p \in \{p_1, \ldots, p_k\}$ and $g \Rightarrow v = \perp$ holds then $\varrho(v) = p$ else $\varrho(v) = v$. Moreover, $\varrho(w) = v$ implies $w = v$. We say that $\mathcal{S}$ only records the first and last occurrence of parameters if, whenever $\varrho(v) = p$ in some transition, v either records the first or the last occurrence of p.

Example 6.2 The login system in Figure 1.3 is a restricted scalarset Mealy machine, because the system only records the first ID and password entered in a Register input. Accordingly, it is not possible to change the login credentials later. □

6.1.2 Abstractions for Restricted SMMs

For each event signature, we introduce a family of symbolic abstractions, parametrized by what we call an *abstraction table*. For each parameter p, an abstraction table contains a list of variables and constants. If v occurs in the list for p then, intuitively, this means that for the future behavior of the SUT it may be relevant whether p equals v or not.

Definition 6.3 (Abstraction table) Let $\Sigma = \langle T_I, T_O \rangle$ be an event signature and let P and U be the sets of parameters that occur in T_I and T_O, respectively. For each $p \in P$, let v_p^f and v_p^l be fresh variables with $\mathsf{type}(v_p^f) = \mathsf{type}(v_p^l) = \mathsf{type}(p) \cup \{\perp\}$, and let $V^f = \{v_p^f \mid p \in P\}$ and $V^l = \{v_p^l \mid p \in P\}$. An *abstraction table* for Σ is a function $F : P \cup U \to (V^f \cup V^l \cup C)^*$, such that, for each $p \in P \cup U$, all elements of sequence $F(p)$ are distinct, and, for each $p \in U$, $F(p)$ lists all the elements of $V^f \cup V^l \cup C$.

$\mathsf{Full}(\Sigma)$ is the abstraction table, where, in addition, for each $p \in P$, $F(p)$ lists all the elements of $V^f \cup V^l \cup C$. Each abstraction table F induces a mapper. This mapper records, for each parameter p, the first and last value of this parameter in a run, using variables v_p^f and v_p^l, respectively. In order to compute the abstract value for a given concrete value d for a parameter p, the mapper checks for the first variable or constant in sequence $F(p)$ with value d. If there is such a variable or constant, the mapper returns the index in $F(p)$, otherwise it returns $\perp$.

Definition 6.4 (Mapper induced by abstr. table) Let $\Sigma = \langle T_I, T_O \rangle$ be a signature and let F be an abstraction table for Σ. Let P be the set of parameters in T_I and let U be the set of parameters in T_O. Let, for $p \in P \cup U$, p' be a fresh variable with $\mathsf{type}(p') = \{0, \ldots, |F(p)| - 1\} \cup \{\perp\}$. Let $T_X = \{\varepsilon(p'_1, \ldots, p'_k) \mid \varepsilon(p_1, \ldots, p_k) \in T_I\}$ and $T_Y = \{\varepsilon(p'_1, \ldots, p'_l) \mid \varepsilon(p_1, \ldots, p_l) \in T_O\}$. Then the map-

per $\mathcal{A}^F_\Sigma = \langle I, O, X, Y, R, r^0, \delta, \mathit{abstr}\rangle$ is defined as follows:

- $I = [\![T_I]\!]$, $O = [\![T_O]\!]$, $X = [\![T_X]\!]$, and $Y = [\![T_Y]\!]$.
- $R = \mathsf{Val}(V^f \cup V^l)$ and $r^0(v) = \perp$, for all $v \in V^f \cup V^l$.
- $\rightarrow$ and abstr are defined as follows, for all $r \in R$,
  1. Let $o = \varepsilon_O(d_1, \ldots, d_k)$ and let $\varepsilon_O(q_1, \ldots, q_k) \in T_O$. Then $r \xrightarrow{o} r$ and $\mathit{abstr}(r, o) = \varepsilon_O(\mathit{first}([\![F(q_1)]\!]_r, d_1), \ldots, \mathit{first}([\![F(q_k)]\!]_r, d_k))$, where for a sequence of values σ and a value d, $\mathit{first}(\sigma, d)$ equals $\perp$ if d does not occur in σ, and equals the smallest index m with $\sigma_m = d$ otherwise, and for a sequence of terms $\rho = t_1 \cdots t_n$ and valuation ξ, $[\![\rho]\!]_\xi = [\![t_1]\!]_\xi \cdots [\![t_n]\!]_\xi$.
  2. Let $i = \varepsilon_I(d_1, \ldots, d_k)$, $\varepsilon_I(p_1, \ldots, p_k) \in T_I$, $r_0 = r$ and, for $1 \leq j \leq k$,

$$r_j \quad = \quad \begin{cases} r_{j-1}[v^f_{p_j} := d_j][v^l_{p_j} := d_j] & \text{if } r_{j-1}(v^f_{p_j}) = \perp \\ r_{j-1}[v^l_{p_j} := d_j] & \text{otherwise} \end{cases} \tag{1}$$

  Then $r \xrightarrow{i} r_k$ and $\mathit{abstr}(r, i) = \varepsilon_I(d'_1, \ldots, d'_k)$, where, for $1 \leq j \leq k$, $d'_j = \mathit{first}([\![F(p_j)]\!]_{r_j - 1}, d_j)$.

Strictly speaking, the mappers $\mathcal{A}^F_\Sigma$ introduced above are not output-predicting: in each state r of the mapper there are infinitely many concrete outputs that are mapped to the abstract output $\perp$. However, in SUTs whose behavior can be described by scalarset Mealy machines, the only possible values for output parameters are constants and values of previously received inputs. As a result, the mapper will never send an abstract output with a parameter $\perp$ to the *learner*. This in turn implies that in the deterministic hypothesis $\mathcal{H}$ generated by the *learner*, $\perp$ will not occur as an output parameter. (Hypotheses in LearnLib only contain outputs actions that have been observed in some experiment.) Since $\mathcal{A}^F_\Sigma$ is output-predicting for all the other outputs, it follows by Lemma 3.4 that the concretization $\gamma_{\mathcal{A}^F_\Sigma}(\mathcal{H})$ is deterministic.

The two theorems below solve (at least in theory) the problem of learning a deterministic symbolic Mealy machine $\mathcal{S}$ that only records the first and last occurrence of parameters. By Theorems 6.1 and 6.2, we know that $\mathcal{M} = \alpha_{\mathcal{A}^{\mathsf{Full}(\Sigma)}_\Sigma}([\![\mathcal{S}]\!])$ is finitary and behavior deterministic. Thus we may apply the approach described in Section 3.1.5 with mapper $\mathcal{A}^{\mathsf{Full}(\Sigma)}_\Sigma$ in combination with any tool that is able to learn finite deterministic Mealy machines. The only problem is that in practice the state-space of $\mathcal{M}$ is too large, and beyond what state-of-the-art learning tools can handle. The proofs of Theorems 6.1 and 6.2 can be found in [57, 3]. They exploit the symmetry that is present in SMMs: using constant preserving automorphisms [89] we exhibit a finite bisimulation quotient and behavior determinacy.

Theorem 6.1 Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma\rangle$ be a SMM that only records the first and last occurrence of parameters. Let F be an abstraction table for Σ. Then $\alpha_{\mathcal{A}^F_\Sigma}([\![\mathcal{S}]\!])$ is finitary.

Theorem 6.2 Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma\rangle$ be a deterministic SMM that only records the first and last occurrence of parameters. Then $\alpha_{\mathcal{A}^{\mathsf{Full}(\Sigma)}_\Sigma}([\![\mathcal{S}]\!])$ is behavior deterministic.

Example 6.3 Consider again our example of a login procedure in Figure 1.3. The mapper induced by the full abstraction table has 8 state variables, which record the first and last values of 4 parameters. This means that for each parameter there are 9 abstract values. Hence, for each of the event primitives Login and Register, we need 81 abstract input actions. Including the Logout input we need 163 abstract inputs. The performance of LearnLib degrades severely if the number of inputs exceeds 20, and learning models with 163 inputs typically is not possible.

However, we can define a mapper $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, abstr \rangle$ for the login system with less abstract inputs. The sets I and O of the mapper are equivalent to T_I and T_O of $\mathcal{S}$. The mapper records the login name and password selected by the user: $R = (\mathbb{N} \cup \{\bot\}) \times (\mathbb{N} \cup \{\bot\})$. Initially, no login name and password have been selected: $r_0 = (\bot, \bot)$. The state of the mapper only changes when a Register input occurs in the initial state:

$$\delta((i,p),a) \quad = \quad \begin{cases} (i',p') & \text{if } (i,p) = (\bot,\bot) \wedge a = \mathsf{Register}(i',p') \\ (i,p) & \text{if } (i,p) \neq (\bot,\bot) \vee a \notin \{\mathsf{Register}(i',p') \mid i',p' \in \mathbb{N}\}. \end{cases}$$

The abstraction forgets the parameters of the input actions, and only records whether a login is correct or wrong: $X = \{\mathsf{Register}, \mathsf{CLogin}, \mathsf{WLogin}, \mathsf{Logout}\}$ and $Y = O$. The abstraction function *abstr* is defined in the obvious way, the only interesting case is the Login input:

$$abstr((i,p), \mathsf{Login}(i',p')) \quad = \quad \begin{cases} \mathsf{CLogin} & \text{if } (i,p) = (i',p') \\ \mathsf{WLogin} & \text{otherwise} \end{cases}$$

Mapper $\mathcal{A}$ is output predicting since *abstr* acts as the identity function on outputs.

This mapper contains an optimal abstraction with just 4 inputs. In the next section, we present a CEGAR approach that allows us to infer an abstraction with 7 inputs. □

6.2 Counterexample-Guided Abstraction Refinement

In order to avoid the practical problems that arise with the abstraction table $\mathsf{Full}(\Sigma)$, we take an approach based on counterexample-guided abstraction. We start with the simplest mapper, which is induced by the abstraction table F with $F(p) = \epsilon$, for all $p \in P$, and only refine the abstraction (i.e., add an element to the table) when we have to. For any table F, $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is finitary by Theorem 6.1. If, moreover, $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is behavior deterministic then LearnLib can find a correct hypothesis and we are done. Otherwise, we refine the abstraction by adding an entry to our table. Since there are only finitely many possible abstractions and the abstraction that corresponds to the full table is behavior deterministic, by Theorem 6.2, our CEGAR approach will always terminate.

During the construction of a hypothesis we will not observe nondeterministic behavior, even when table F is not full: in Tomte the mapper always chooses a fresh concrete value whenever it receives an abstract action with parameter value $\bot$, i.e. the mapper induced by F will behave exactly as the mapper induced by

$\mathsf{Full}(\Sigma)$, except that the set of abstract actions is smaller. In contrast, during the testing phase Tomte selects random values from a small domain. In this way, we ensure that the full concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is explored. If the *teacher* responds with a counterexample (u, s), with $u = i_1, \ldots, i_n$ and $s = o_1, \ldots, o_n$, we may face a problem: the counterexample may be due to the fact that $\mathcal{H}$ is incorrect, but it may also be due to the fact that $\alpha_{\mathcal{A}^F_\Sigma}([\![\mathcal{S}]\!])$ is not behavior-deterministic. In order to figure out the nature of the counterexample, we first construct the unique execution of $\mathcal{A}^F_\Sigma$ with trace $i_1 o_1 i_2 o_2 \cdots i_n o_n$. Then we assign a color to each occurrence of a parameter value in this execution:

Definition 6.5 Let $r \xrightarrow{i} r'$ be a transition of $\mathcal{A}^F_\Sigma$ with $i = \varepsilon_I(d_1, \ldots, d_k)$ and let $\varepsilon_I(p_1, \ldots, p_k) \in T_I$. Let $abstr(r, i) = \varepsilon_I(d'_1, \ldots, d'_k)$. Then we say that the occurrence of value d_j is *green* if $d'_j \neq \perp$. Occurrence of value d_j is *black* if $d'_j = \perp$ and d_j equals the value of some constant or occurs in the codomain of state r_{j-1} (where r_{j-1} is defined as in equation (1) above). Occurrence of value d_j is *red* if it is neither green nor black.

Intuitively, an occurrence of a value of an input parameter p is green if it equals a value of a previous parameter or constant that is listed in the abstraction table, an occurrence is black if it equals a previous value that is not listed in the abstraction table, and an occurrence is red if it is fresh. The mapper now does a new experiment on the SUT in which all the black occurrences of input parameters in the trace are converted into fresh "red" occurrences. If, after abstraction, the trace of the original counterexample and the outcome of the new experiment are the same, then hypothesis $\mathcal{H}$ is incorrect and we forward the abstract counterexample to the *learner*. But if they are different then we may conclude that $\alpha_{\mathcal{A}^F_\Sigma}(\mathcal{S})$ is not behavior-deterministic and the current abstraction is too coarse. In this case, the original counterexample contains at least one black occurrence, which determines a new entry that we need to add to the abstraction table.

The procedure for finding this new abstraction is outlined in Algorithm 6.1. Here, for an occurrence b, $\mathsf{param}(b)$ gives the corresponding formal parameter, $\mathsf{source}(b)$ gives the previous occurrence b' which, according to the execution of $\mathcal{A}^F_\Sigma$, is the source of the value of b, and $\mathsf{variable}(b)$ gives the variable in which the value of b is stored in the execution of $\mathcal{A}^F_\Sigma$. To keep the presentation simple, we assume here that the set of constants is empty. If changing some black value b into a fresh value changes the observable output of the SUT, and also a change of $\mathsf{source}(b)$ into a fresh value leads to a change of the observable output, then this strongly suggests that it is relevant for the behavior of the SUT whether or not b and $\mathsf{source}(b)$ are equal, and we obtain a new entry for the abstraction table. If changing the value of either b or $\mathsf{source}(b)$ does not change the output, we obtain a counterexample with fewer black values. If b is the only black value then, due to the inherent symmetry of SMMs, changing b or $\mathsf{source}(b)$ to a fresh value in both cases leads to a change of observable output. When the new abstraction entry has been added to the abstraction table, the *learner* is restarted with the new abstract alphabet.

Algorithm 6.1 Abstraction refinement

Input: Counterexample $c = i_1 \cdots i_n$
Output: Pair (p, v) with v new entry for $F(p)$ in abstraction table
Function refineAbstraction(c)
1: **while** abstraction not found **do**
2: Pick first occurrence of a black value b from c
3: $c' := c$, where param(b) is set to a fresh value f
4: **if** output $s'[f/b]$ from running c' on SUT $\neq$ output of c **then**
5: $c'' := c$, where param(source(b)) is set to a fresh value f
6: **if** output $s''[f/b]$ from running c'' on SUT $\neq$ output of c **then**
7: **return** (param(b), variable(source(b)))
8: **else** $c := c''$
9: **end if**
10: **else** $c := c'$
11: **end if**
12: **end while**

Note: $[f/b]$ denotes the substitution of b by f

6.3 Example Applications

We have implemented our approach in the Tomte tool to infer models of realistic systems that can be represented as a scalarset Mealy machine. In this section, we illustrate the different systems and discuss for several of them how abstractions of the input and output have been learned automatically.

6.3.1 Login Procedure

The scalarset Mealy machine of the login procedure is depicted in Figure 1.3. The input alphabet of the *learner* is initialized with the following abstract symbols: Register($\perp, \perp$) and Login($\perp, \perp$) and Logout(), where $\perp$ is the 'default' abstract value. There is no abstract value different from $\perp$ and thus, in the beginning of our algorithm all values of a parameter are in one large equivalence class. The mapper is equipped with eight state variables ($firstRegisterId$, $lastRegisterId$, $firstRegisterPw$, etc.) storing the first and last occurrence of the parameters in the Register and Login messages. The *learner* starts by asking output queries using the three abstract symbols. For output queries the mapper always selects the smallest fresh value for every $\perp$, e.g. the abstract output query Register($\perp, \perp$) Login($\perp, \perp$) is translated to Register($1, 2$) Login($3, 4$) if the start value is set to 1. In this way the *learner* constructs the abstract hypothesis shown in Figure 6.1. Since the mapper only uses fresh values in the output queries, the hypothesis does not contain the branch, where the login is successful.

To test whether the hypothesis is correct, the *learner* asks an inclusion query. Now the mapper behaves in a different way to concretize inputs. Instead of choosing fresh values for every $\perp$, the mapper selects random values from a small range.

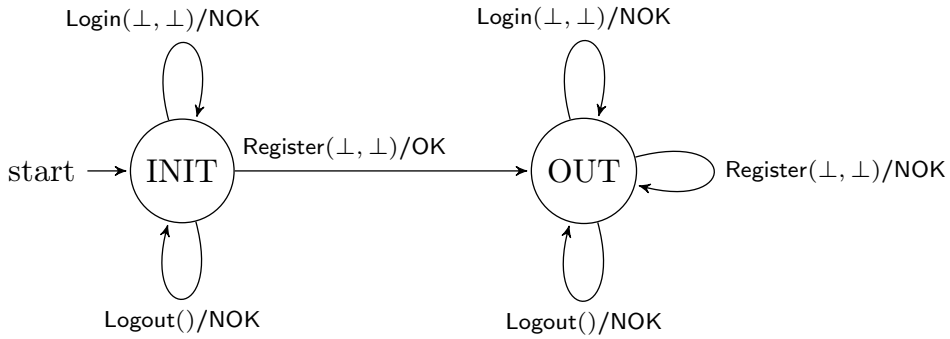


Figure 6.1: *First hypothesis of login procedure*

For example, consider the abstract test trace Register followed by two Login inputs, which is translated to a concrete trace with many duplicate values, see line 2 below.

1:	abstract inputs:	Register$(\perp, \perp)$	Login$(\perp, \perp)$	Login$(\perp, \perp)$
2:	concrete inputs:	Register$(3, 17)$	Login$(17, 5)$	Login$(3, 17)$
3:	outputs SUT:	OK	NOK	OK
4:	outputs hypothesis:	OK	NOK	NOK

The output of this trace is OK (line 3), because the 3 and 17 of the last Login are equivalent to the ID and password of the first Register message. This is different from the output produced by the hypothesis, which generates a NOK (line 4). Thus, we have found a counterexample. Now there are two possibilities. Either the *learner* has not found all states and we need to forward the abstract counterexample to the *learner* to solve the problem, or our current abstraction is too coarse and we need to refine it. To determine which case it is, we convert the input sequence of the counterexample into a fresh trace by replacing all $\perp$ values with a fresh value, see line 5 below, and rerun it on the SUT. For this trace we know that all duplicate values (if present) are already covered by existing abstractions and, therefore, no additional abstractions can resolve the nondeterminism. Thus, if it is still a counterexample (line 6), only the *learner* can solve it. If we take a look at the SUT in Figure 1.3, the new experiment results in a NOK output (line 7). So we have to refine our abstraction.

5:	fresh trace:	Register$(1, 2)$	Login$(3, 4)$	Login$(5, 6)$
6:	counterexample for *learner*:	OK	NOK	OK
7:	abstraction refinement:	OK	NOK	NOK

For this purpose, we identify the green and black values in the trace, see line 9 below, where values 3 and 17 are black, and no values are green. Then we try to remove black values, because they refer to possible abstractions. We start with the first black value, i.e. the 17 in the first Login and replace it with a fresh value, e.g. with value 4, see line 10. We rerun the trace on the SUT and observe the same output as before (line 11). Accordingly, value 17 in the first Login input was not relevant for the behavior of the SUT and so we can leave fresh value 4 in the trace.

Now we try the same with the second black value: value 3 in the second Login input. Again, we replace the black value with a fresh value, e.g. 1 (line 12) and rerun the trace on the SUT. This time we observe a different output, see line 13. Because abstractions always exist between two values, we need to verify that the source of the black value is also relevant. We reset the black value and introduce a fresh value for the first 3 in the Register message (line 14). Running the new trace on the SUT gives again a different output than produced by the original counterexample, see line 15. Thus, we have found a new abstraction. This new abstraction refers to the equality between the Login ID parameter (id1 in Figure 1.3) and the $firstRegisterId$ state variable that records the ID of the first Register message.

8:	abstract inputs:	Register$(\perp, \perp)$	Login$(\perp, \perp)$	Login$(\perp, \perp)$
9:	**black** and red values:	Register$(3, 17)$	Login$(\mathbf{17}, 5)$	Login$(\mathbf{3}, \mathbf{17})$
10:	**black value** $\rightarrow$ fresh value:	Register$(3, 17)$	Login$(\underline{4}, 5)$	Login$(\mathbf{3}, \mathbf{17})$
11:	outputs SUT:	OK	NOK	OK
12:	**black value** $\rightarrow$ fresh value:	Register$(3, 17)$	Login$(4, 5)$	Login$(\underline{1}, \mathbf{17})$
13:	outputs SUT:	OK	NOK	NOK
14:	source value $\rightarrow$ fresh value:	Register$(\underline{2}, 17)$	Login$(4, 5)$	Login$(\mathbf{3}, \mathbf{17})$
15:	outputs SUT:	OK	NOK	NOK

All abstractions are stored in an abstraction table, which initially is empty. For our new abstraction, we add the $firstRegisterId$ state variable to the abstractions of the Login ID parameter, see Table 6.1. Moreover, we need to update the abs-

Parameter	**Abstraction(s)**
Register ID ($id0$)	
Register PW ($pw0$)	
Login ID ($id1$)	$firstRegisterId$
Login PW ($pw1$)	

Table 6.1: *Abstraction table for login procedure*

tract alphabet of the *learner* by adding the new symbol Login$(firstRegisterId, \perp)$. With this alphabet the entire learning process is restarted from scratch. During hypothesis construction the mapper copies for the abstract parameter value $firstRegisterId$ in a Login(firstRegisterId, $\perp$) input the value from the $firstRegisterId$ state variable instead of selecting a fresh value. Apart from that, we proceed in the usual way. The next abstraction our CEGAR algorithm will find is the equality between the Login PW parameter and the $firstRegisterPw$ state variable.

6.3.2 Session Initiation Protocol

We also illustrate the operation of Tomte by means of the session initiation protocol (SIP) as presented in Section 3.4.1. Initially, no abstraction for the input is

defined in the *learner*, which means all parameter values are ⊥. As a result every parameter in every input action is treated in the same way and the mapper selects a fresh concrete value, e.g. the abstract input trace *INVITE*(⊥, ⊥, ⊥) *ACK*(⊥, ⊥, ⊥) *PRACK*(⊥, ⊥, ⊥) *PRACK*(⊥, ⊥, ⊥) is translated to the concrete trace *INVITE(1, 2, 3) ACK(4, 5, 6) PRACK(7, 8, 9) PRACK(10, 11, 12).* In the learning phase queries with distinct parameter values are sent to the SUT, so that the *learner* constructs the abstract Mealy machine shown in Figure 6.2. In the testing

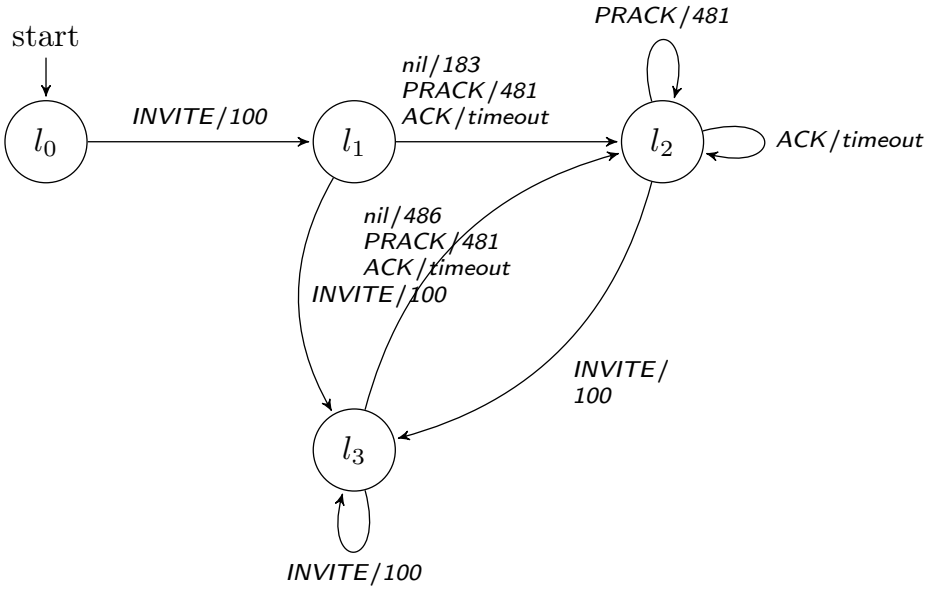


Figure 6.2: *First hypothesis of the SIP protocol. Due to space limitations, we have suppressed the (abstract) parameter values. Moreover, for readability, we have merged transitions with same source and next location and have removed* nil/timeout *transitions.*

phase parameter values may be duplicated, which may lead to nondeterministic behavior. The test trace *INVITE ACK PRACK* below leads to a *200* output that is not foreseen by the hypothesis, which produces a *481*.

Abstract inputs:	*INVITE*(⊥, ⊥, ⊥)	*ACK*(⊥, ⊥, ⊥)	*PRACK*(⊥, ⊥, ⊥)
Concrete inputs:	*INVITE*(16, 17, 9)	*ACK*(**9**, 3, 22)	*PRACK*(**16**, 15, 21)
Outputs SUT:	*100*	*timeout*	*200*
Outputs hypothesis:	*100*	*timeout*	*481*

Rerunning the trace with distinct values as before leads to a *481* output. Thus, to resolve this problem, we need to refine the input abstraction. Therefore, we identify the green and black values in the trace and try to remove black values. The algorithm first replaces the **9** in the *ACK* input with a fresh value and observes the same output as before. However, replacing the **16** in the *PRACK* input with a fresh value changes the final outcome of the SUT to a *481* output. Also replacing the first 16 with a fresh value gives a *481* output. As a result, we refine the input abstraction by adding an equality check between the first parameter of the first *INVITE* message and the first parameter of a *PRACK* message to every *PRACK* input. Apart from refining the input alphabet, every concrete output parameter

value is abstracted to either a constant or a state variable storing the first or last occurrence of a parameter value. After every input abstraction refinement, the learning process is restarted. We proceed until the *learner* finishes the inference process without getting interrupted by a nondeterministic output.

6.3.3 Other SUTs

We also used the Tomte tool to automatically infer guard statements and models of other SUTs. In addition to the login system and session initiation protocol discussed in the previous sections, we applied our approach to the biometric passport as presented in Chapter 4. For the other systems we used the Uppaal [24] GUI as an editor to create an extended finite state machine that models the behavior of the SUT. In our EFSM input symbols start with I and output symbols start with O. In addition to symbols, the transitions may contain value checks (or guards, ==) and assignments (=). By means of several Python scripts we automatically generated from the Uppaal .xml file a Java executable that represents the EFSM as a Java state machine and acts as SUT. In the following paragraphs we give an overall overview of the different systems.

Alternating bit protocol The alternating bit protocol (ABP) is a simple communication protocol for reliable transmission of messages (frames) [23]. Reliability is guaranteed by retransmitting lost or corrupted messages and ensuring the order of messages. The protocol consists of a sender and a receiver, which communicate via two channels: channel M for messages from sender to receiver and channel A for acknowledgements from receiver to sender, see Figure 6.3. Each message the sender transmits contains a data part and an additional bit, i.e., a value that is 0 or 1. After transmission, the sender waits for an acknowledgement from the receiver (with the same bit) via A. If no correct acknowledgement arrives, the sender retransmits the message with the same bit. Otherwise, the sender flips the bit and starts transmitting the next message. The Uppaal state machines we created of the sender, channel M and the receiver are depicted in Appendix 6.A.

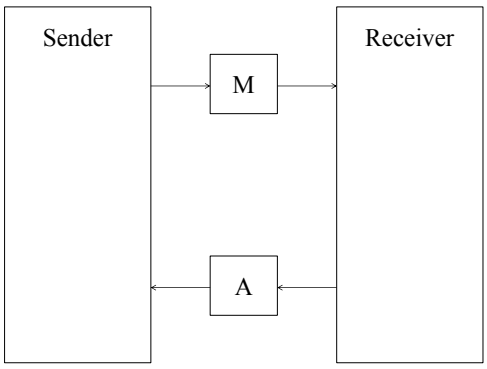


Figure 6.3: *Alternating bit protocol*

River crossing puzzle A farmer has to cross a river with a wolf, a goat, and a cabbage without creating a situation where either goat and wolf or goat and cabbage are at one bank of the river while the farmer is at the other one. The boat is small and the farmer can only transport either the wolf, goat or cabbage when crossing the river, or cross the river alone. There exist different versions of this transport puzzle, e.g. the objects to carry from one side of the river to the other might vary or the river might be replaced by a bridge. Our version of the

river crossing puzzle is shown in Figure 6.7. The input IIN contains a parameter input that can take on the values farmer, wolf, goat, or cabbage, representing attempts to cross the river by respectively farmer alone, or farmer with wolf, goat, or cabbage. When an unsafe situation is created in which one item can eat another one, the output OEATEN() is generated, which returns to the initial state. When an invalid choice is made (an item is chosen that is at the wrong bank of the river), the output ONOK() is returned. When all items successfully crossed the river, the output ODONE() is generated.

Palindrome/repnumber checker The palindrome/repnumber checker depicted in Figure 6.8 can perform two kinds of tests. One can test whether two, three, or four numbers entered are a palindrome or a repnumber. In our case, a palindrome is a sequence of numbers that remains the same when its numbers are reversed. For example, for the inputs IPalindrome3$(12, 4, 12)$ and IPalindrome4$(1, 3, 3, 1)$ output OYes will be returned. A repnumber is composed of repeated instances of the same number. The word repnumber is a portmanteau, formed from repeated number. For example, the inputs IRepnumber2$(20, 20)$, IRepnumber3$(7, 7, 7)$, or IRepnumber4$(5, 5, 5, 5)$ lead to an OYes output.

6.4 Experimental Results

Table 6.2 gives an overview of the systems we learned with the numbers of constant and action parameters used in the models, the number of input refinement steps, total numbers of learning and testing queries (sequences of inputs), number of states of the learned abstract model, and the time needed for learning and testing (in seconds). These numbers and times do not include the last equivalence query, in which no counterexample has been found. In all our experiments, correctness of hypotheses was tested using random walk testing. The outcomes depend on the return value of function variable(b) in case b is the first occurrence of a parameter p: v_p^f or v_p^l. Table 6.2 is based on the optimal choice, which equals v_p^f for SIP and the login procedure, and v_p^l for all the other benchmarks. The biometric passport case study of Chapter 4 has also been learned fully automatically by [83]. All other benchmarks require history dependent abstractions, and Tomte is the first tool that has been able to learn these models fully automatically. We have checked that all models inferred are observation equivalent to the corresponding SUT. For this purpose we combined the learned model with the abstraction and used the CADP tool set, `http://www.inrialpes.fr/vasy/cadp/`, for equivalence checking. Our tool and all models can be found at `http://www.tomte.cs.ru.nl/`.

System under test	Constants/ parameters	Input refine- ments	Learning/ testing queries	States	Learning/ testing time
Alternating bit protocol sender	2/2	1	193/4	7	0.6s/0.1s
Alternating bit protocol receiver	2/2	2	145/3	4	0.4s/0.2s
Alternating bit protocol channel	0/2	0	31/0	2	0.1s/0.0s
Biometric passport (Chapter 4)	3/1	3	2199/2607	5	3.9s/32.0s
Session initiation protocol (Section 3.4.1)	0/3	2	1153/101	14	3.0s/0.9s
Login procedure (Example 1.1)	0/4	2	283/40	4	0.5s/0.7s
River crossing puzzle	4/1	4	610/1279	9	1.7s/16.2s
Palindrome/repnumber checker	0/16	9	1941/126	1	2.4s/3.3s

Table 6.2: *Learning statistics. For the learning (output) and testing queries sequences of inputs (separated by resets) have been counted. Also note that these numbers and times do not include the last equivalence query, in which no counterexample has been found.*

6.A Alternating Bit Protocol

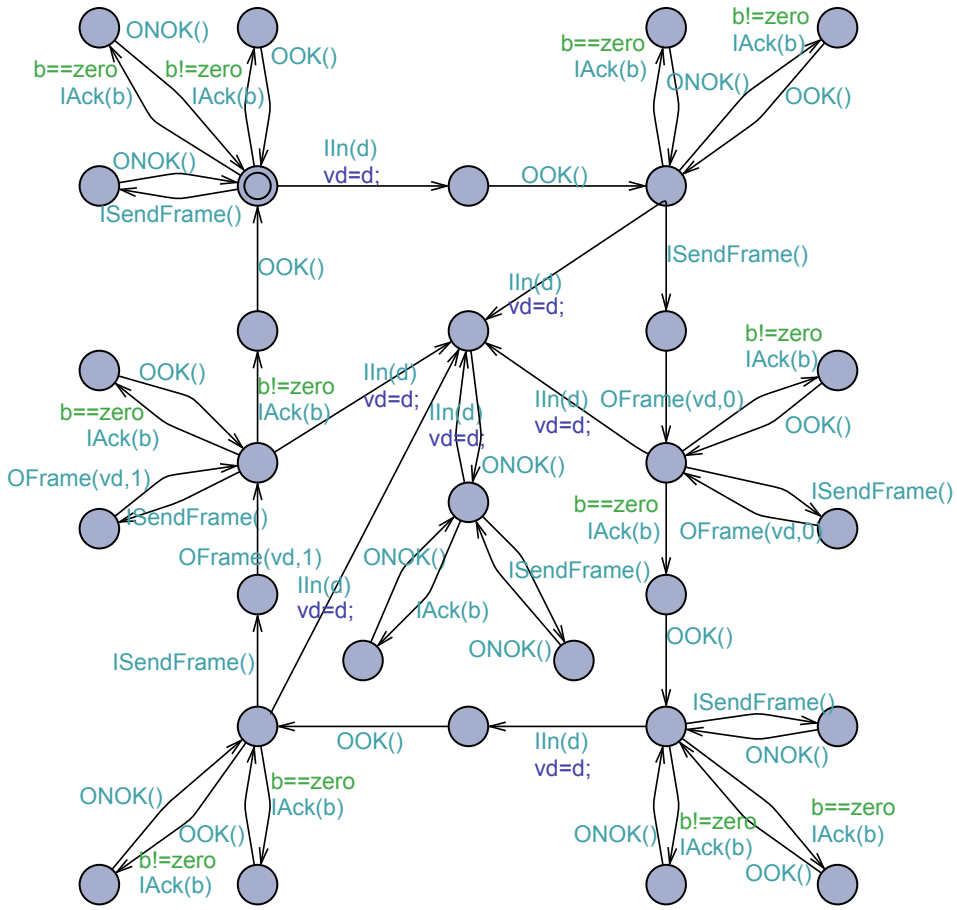


Figure 6.4: *Extended finite state machine model of the alternating bit protocol sender*

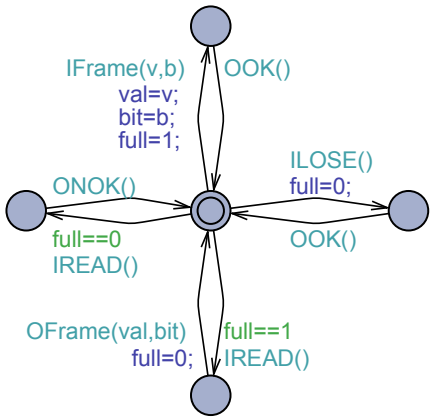


Figure 6.5: *Extended finite state machine model of the alternating bit protocol channel for transmission of messages (frames)*

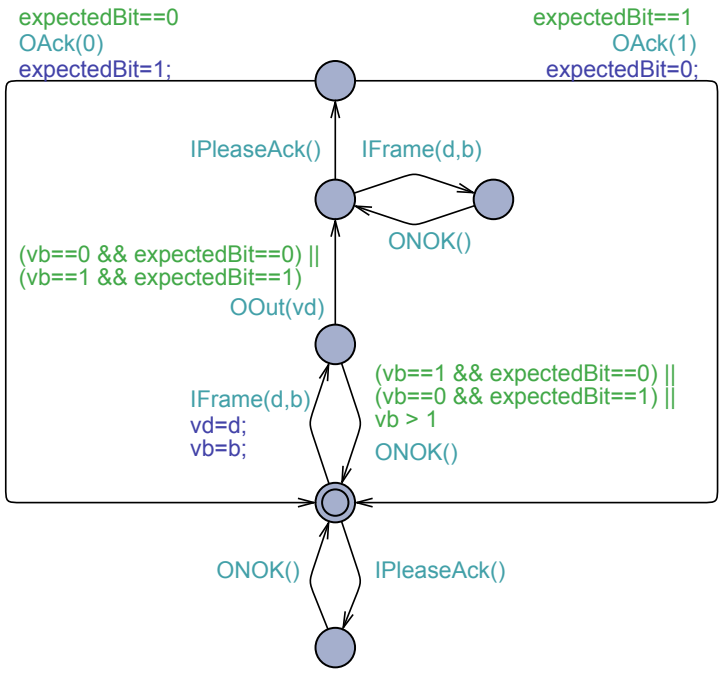


Figure 6.6: *Extended finite state machine model of the alternating bit protocol receiver*

6.B River Crossing Puzzle

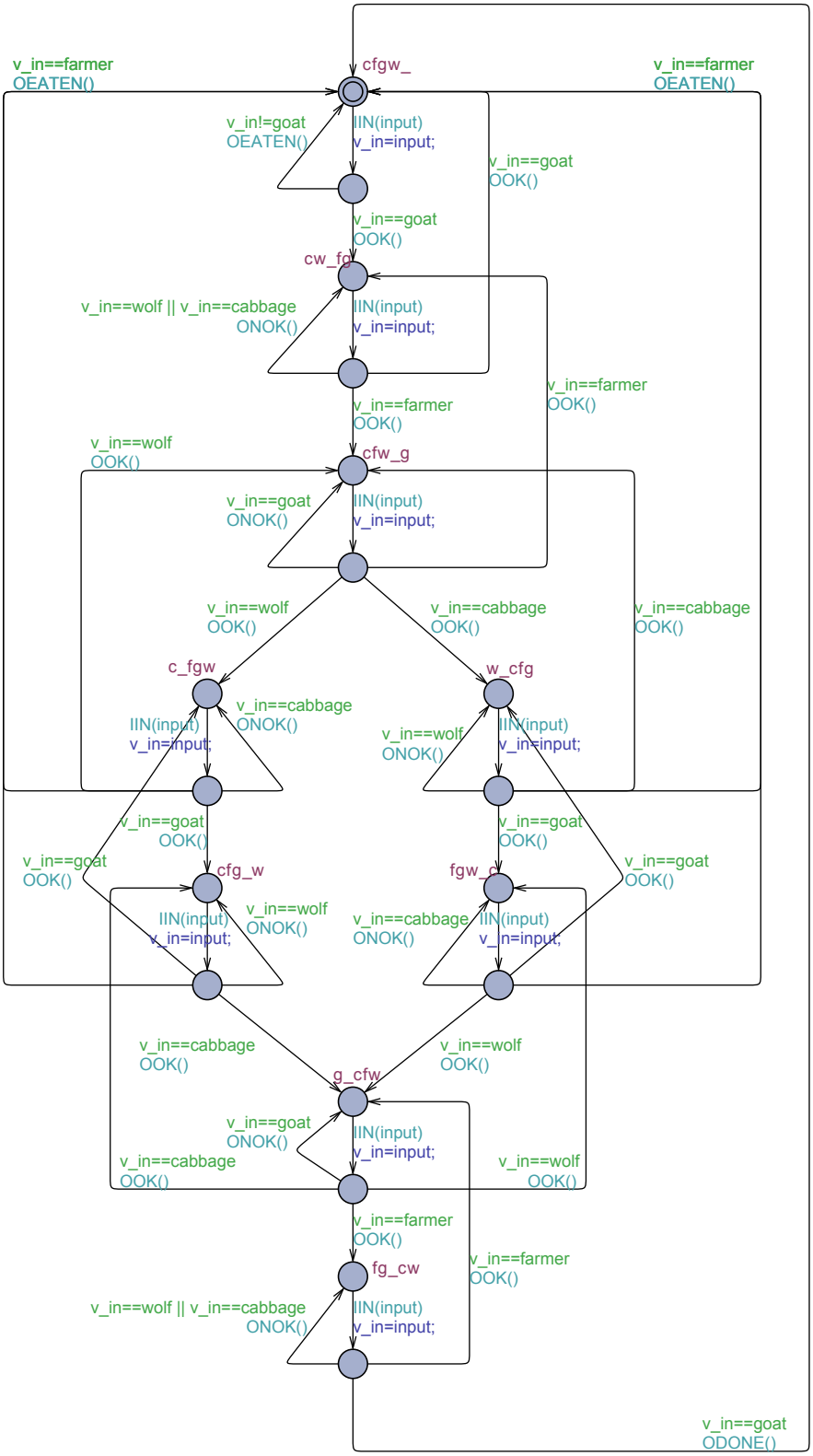


Figure 6.7: *Extended finite state machine model of the river crossing puzzle*

6.C Palindrome/Repnumber Checker

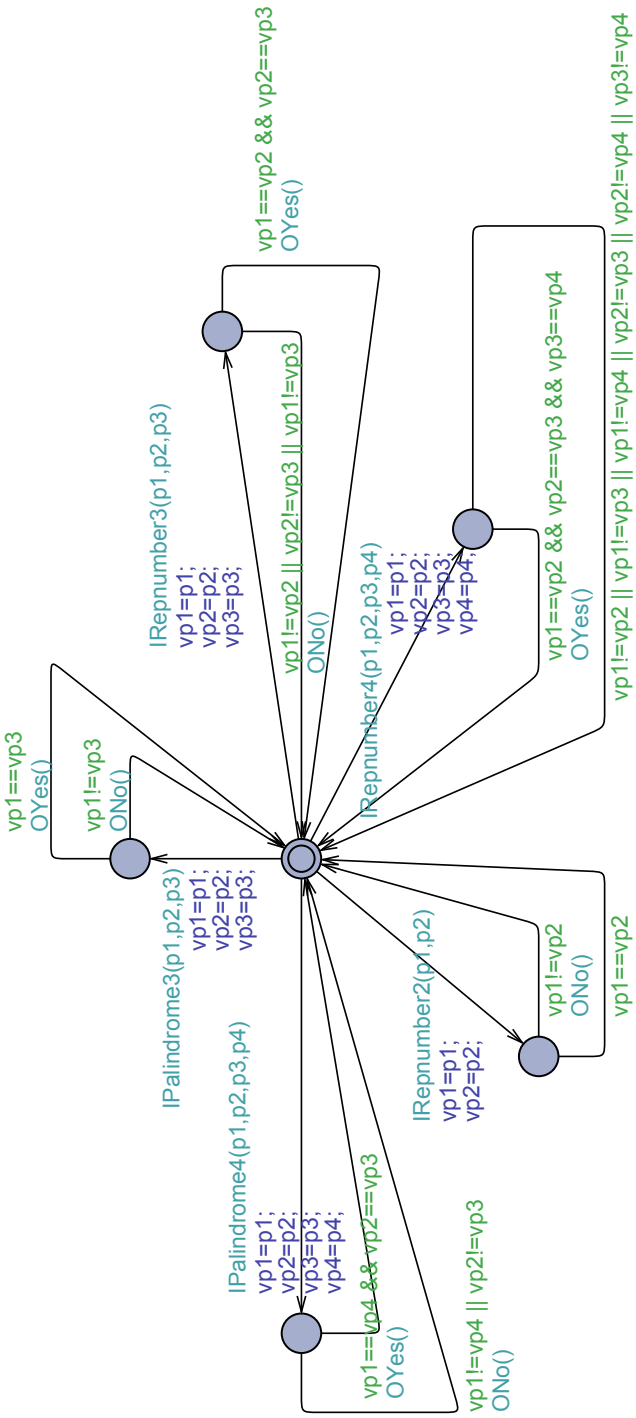


Figure 6.8: *Extended finite state machine model of the palindrome/repnumber checker*

Chapter 7
Active Learning using a Lookahead Oracle

In the previous chapter, we have presented an approach based on counterexample-guided abstraction refinement (CEGAR) to learn models of SUTs that can be defined using deterministic scalarset Mealy machines (SMM). These SMM can test for equality between data parameters or between a parameter and a constant, but no operations on data are allowed. Moreover, these SMM can only record the first and last occurrence of a parameter. In this chapter, we relax the last restriction and, accordingly, extend the class of systems we can learn.
The idea in Chapter 6 was that the mapper records, for each parameter p, the first and last value of this parameter in a run, using variables v_p^f and v_p^l, respectively. This approach works fine for certain communication protocols which remember the first value being selected to configure the settings, e.g. the identifier of the connection to establish. Communication protocols often also remember the values received in the most recent input message to produce the corresponding reply, but thereafter forget them. However, there are many systems for which the approach of Chapter 6 does not work.

Example 7.1 Consider an SMM that models a stack of capacity three, see Figure 7.1. If the stack is not full, one can store an additional value in the data structure

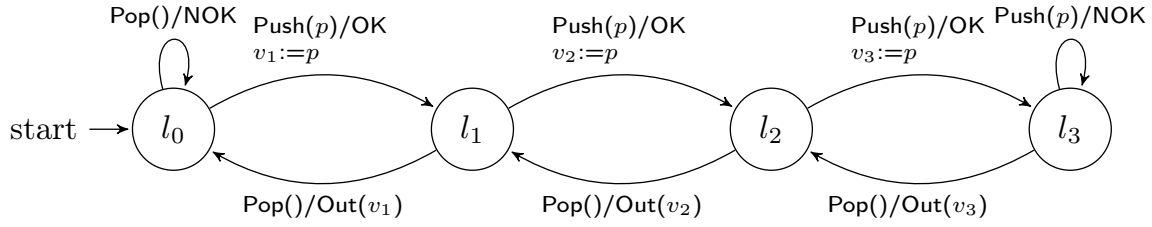


Figure 7.1: *Stack with a capacity of 3 modeled as a scalarset Mealy machine*

using a Push(p) input. If the stack is not empty, one can retrieve the last inserted value by means of a Pop() input. A problem with the current approach arises when we push three elements on the stack and then perform two Pop() inputs. The first value is stored in the v_p^f variable, the last (third) value in the v_p^l variable, and all intermediate values (second value) are not remembered. □

In this chapter, instead of using a predefined set of state variables, we generalize our approach such that it will automatically detect the state variables needed. For this purpose, we introduce the concept of *memorable values*, similar to [80, 79].

7.1 Memorable Values

A value is *memorable* if it has to be remembered, because it will be used later.

Example 7.2 In the stack of Example 7.1 there are at most three memorable values, i.e. the three values that are stored in the data structure after three $\mathsf{Push}(p)$ inputs have been performed. To figure out that there are at least three memorable values, we have to send three $\mathsf{Pop}()$ inputs after the input sequence $\mathsf{Push}(value1)$ $\mathsf{Push}(value2)$ $\mathsf{Push}(value3)$ with $value1$, $value2$, and $value3$ distinct, which will result in the output sequence $\mathsf{Out}(value3)$ $\mathsf{Out}(value2)$ $\mathsf{Out}(value1)$. □

By looking ahead, we can find out that certain values will be used in the future and therefore need to be memorized.

Example 7.3 Consider again the login procedure of Chapter 6 shown in Figure 1.3. Values can also be *memorable* if they are referred to in a guard statement at a later date. In the login system we need to remember the login name and password selected during the registration. These values will never occur as a parameter of an output action, but they are used to decide whether a login is successful or not. □

Intuitively, a data value after u is memorable if it has an impact on the future behavior, i.e.

1. some continuation of u uses d as a parameter value in an output, or
2. some continuation of u uses d in a guard, which causes that a different output is generated.

More formally, a *memorable value* can be defined as follows:

Definition 7.1 (Memorable value) Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a scalarset Mealy machine, suppose $l_0 \overset{u/s}{\Rightarrow} l$, and let d be a parameter value that occurs in u and that is not denoted by any constant ($\forall c \in C : \gamma(c) \neq d$). Then d is *memorable* after u if there is a witness transition $l \overset{v/t}{\Rightarrow} l'$, where (1) d occurs in output t and not in input v, or (2) d occurs in input v and if we replace all occurrences of d in v with a fresh value f then $l \overset{v[f/d]/t'}{\Rightarrow} l''$ with $t' \neq t[f/d]$.

Let $memV(u)$ be the set of memorable values after u. To derive memorable values, we introduce a new intermediate component called *lookahead oracle*. It functions as a cache and computes memorable values. As in Chapter 6, we place a mapper in between the *learner* and the *teacher* that translates abstract symbols to concrete symbols and vice versa. In addition, we place the lookahead oracle in between the mapper module and the SUT. The overall set-up of the new learning framework is depicted in Figure 7.2.

Let $\mathcal{A}$ be a mapper in the sense of Definition 3.1 with concrete actions $I \cup O$ and abstract actions $X \cup Y$. Then the messages exchanged between the different

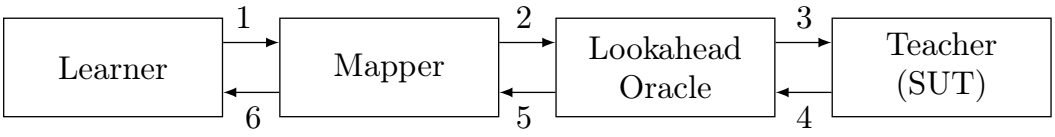


Figure 7.2: *Learning framework*

components during the construction of a hypothesis are as follows (see numbering in Figure 7.2):

1. The *learner* sends an abstract output query $x \in X$ to the mapper.
2. When the mapper receives an abstract query x, it nondeterministically selects a concrete input symbol $i \in I$ and forwards it as an output query to the lookahead oracle. If no such i exists, it returns $\perp$ to the *learner*.
3. The lookahead oracle checks whether it has cached i. If this is not the case, then the lookahead oracle forwards i to the *teacher*. Otherwise, go to Step 5.
4. The *teacher* sends a concrete answer $o \in O$ to the lookahead oracle, which caches the input-output pair i/o.
5. The lookahead oracle returns o together with the memorable values after u, where u is the sequence of output queries sent after the last reset query up to and including i, to the mapper. How the lookahead oracle computes the memorable values will be discussed in Section 7.2.
6. When the mapper receives a concrete answer o and the set of memorable values from the lookahead oracle, it identifies the abstract answer $y \in Y$ and forwards it to the *learner*. This abstract output specifies, amongst others, the update of state variables, determined by means of the memorable values.

From the learner's point of view, we can sum up these six learning steps in a function $\mathsf{runFresh}(x_1 \dots x_n)$, which takes as argument an abstract input sequence and returns an abstract output sequence $y_1 \dots y_n$. The function determines stepwise for every x_j, where $1 \leq j \leq n$, the concrete input i_j, the concrete output o_j and the memorable values after $i_1 \dots i_j$ such that it can compute y_j.

During the verification of a hypothesis all concrete test sequences are forwarded from the mapper straight to the SUT, without traversing the lookahead oracle. Answers from the SUT are also returned straight to the mapper. The reason for this is that test traces are usually very long and the inputs in the beginning of a test trace often do not overlap so that most likely for every test trace a separate branch has to be created in the tree. The huge observation tree gets even more unmanageable if, in addition, all lookahead traces are performed for every node in the tree, which also leads to excessive communication with the SUT.

7.2 Lookahead Oracle

A lookahead oracle is a component that stores traces of an SUT in a so-called *observation tree*. The most obvious advantage of storing the traces is that the

tree functions as a cache for repeated output queries on the SUT. However, a more interesting application is that by means of an observation tree we can detect memorable values by looking ahead in the tree. The details of how this works will be presented in this section.

7.2.1 Observation Tree

An *observation tree* stores concrete traces and tells us, which parameter values need to be recorded at a certain time.

Definition 7.2 (Observation tree) Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a scalarset Mealy machine. Let $\Sigma = \langle T_I, T_O \rangle$, $I = [\![T_I]\!]$, and $O = [\![T_O]\!]$. Then an *observation tree* for $\mathcal{S}$ is a tuple $\mathcal{OT}_\mathcal{S} = \langle \mathcal{N}, N_0, \mathcal{E} \rangle$, where $\mathcal{N}$ is a finite set of nodes, $N_0 \in \mathcal{N}$ is the root node, and $\mathcal{E} \subseteq \mathcal{N} \times I \times O \times \mathcal{N}$ is a set of edges. $\mathcal{OT}_\mathcal{S}$ has a tree-like structure, i.e. every node $N \in \mathcal{N}$, except for N_0, has exactly one incoming edge. We require that for all $N \in \mathcal{N}$ and $i \in I$ there is at most one edge $(N, i, o, N') \in \mathcal{E}$.

An edge $E \in \mathcal{E}$ in $\mathcal{OT}_\mathcal{S}$ is labeled with a concrete input $i \in I$ and output symbol $o \in O$ and connects two nodes $N, N' \in \mathcal{N}$. We write $N \xrightarrow{i/o} N'$ if $(N, i, o, N') \in \mathcal{E}$. At any point in time, an observation tree is in some node $N \in \mathcal{N}$, which we call the *current node*. It is possible to use the tree as a cache by giving concrete inputs to the tree, i.e., by supplying an input symbol $i \in I$. The tree then selects the edge $N \xrightarrow{i/o} N'$, produces output symbol $o \in O$, and updates the current node to the new node N'.

The edges are extended to finite sequences by defining $\overset{u/s}{\Rightarrow}$ to be the least relation that satisfies, for $N, N', N'' \in \mathcal{N}$, $u \in I^*$, $s \in O^*$, $i \in I$, and $o \in O$, if $N \xrightarrow{i/o} N'$ and $N' \overset{u/s}{\Rightarrow} N''$ then $N \overset{i\,u/o\,s}{\Rightarrow} N''$.

Every node N in the observation tree contains a set of memorable values $MemV$. We write $N.MemV$ for the set of memorable values recorded for N. A value in N is *memorable* after u if it will be needed in some continuation of u, i.e. in some edge leading to a successor node of N.

Example 7.4 Assume we want to construct an observation tree for the login procedure in Figure 1.3. We run concrete traces of length 2 on the SUT and store them in the tree as depicted in Figure 7.3. Note that node N_2 has a set with two memorable values {0,1}. The witness sequence Login(0,1) executed after Register(0,1) proves by means of an OK output that both 0 and 1 are memorable since Login(0,2) and Login(2,1) result in a NOK.

□

In order to fill the observation tree with concrete traces or to add memorable values to a node in the tree, some basic operations have been defined on observation trees:

1. **Inserting a node:** The function insertNode(N_p, i) adds a new child node N_c to the node N_p, connected by an edge labeled i/o, where the corresponding concrete output symbol o is unknown. To determine o, the lookahead oracle has to communicate with the SUT. For this purpose, we first reset the SUT to its initial state. Then, we send the input sequence defined by the concrete

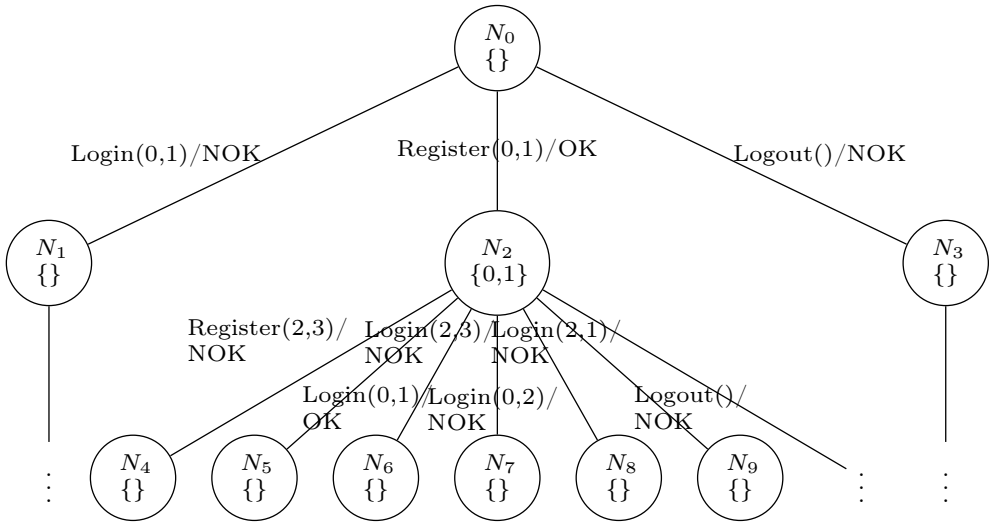


Figure 7.3: *Observation tree of login procedure*

input symbols on the edges from the root node to N_p to the SUT before we send the concrete input i. Using the last input symbol o that is returned by the SUT, we construct the edge i/o that is connected to the new child node N_c. N_c is initialized with an empty set of memorable values.

2. **Running a trace:** The function $\mathsf{run}(u = i_1 \dots i_k,\ v = i_{k+1} \dots i_m)$ starts at the root node and traverses the tree by successively selecting the edge that conforms to the concrete input symbol i in u and v, i.e. $N_0 \xrightarrow{i_1/o_1} N_1 \cdots \xrightarrow{i_k/o_k} N_k \xrightarrow{i_{k+1}/o_{k+1}} N_{k+1} \cdots \xrightarrow{i_m/o_m} N_m$. While processing input sequence v, the function collects all outputs $o_{k+1} \dots o_m$ and, finally, returns them. If some node N_j with $1 \leq j \leq m$ does not have an edge with input i, the according edge and child node are added by calling the function $\mathsf{insertNode}(N_j,\ i)$, see Item 1.

3. **Adding a memorable value:** The function $\mathsf{addMemV}(N,\ D)$ adds a set of values D to the memorable values of node N.

7.2.2 Lookahead Traces

To compute the memorable values for a node N in the observation tree, the lookahead oracle contains a list of so-called *lookahead traces*. These lookahead traces are run starting at N to explore the near future of that node.

Definition 7.3 (Lookahead trace) A lookahead trace lt is a sequence of parameterized input actions of form $\varepsilon(p_1, \dots, p_n)$, where p_i is a pair (type,index) with type $\in$ {'f', 'c', 'l'} and index $\in \mathbb{N}$.

Intuitively, a lookahead trace is an abstract form of a trace, where each parameter refers to either a previous parameter ('l'), or a specified constant ('c'), or it is fresh ('f'). A lookahead trace can be converted into a concrete lookahead trace by replacing each (type,index) pair with a matching concrete parameter value. Sometimes, several matches are possible for a parameter of type 'l', leading to more

than one concrete lookahead trace. If a specific pair occurs multiple times in a lookahead trace, it will be replaced multiple times with the same concrete value in the concrete lookahead trace. These concrete lookahead traces can then be run on the SUT. We maintain two lists of lookahead traces in the lookahead oracle: the list $\mathcal{OLT}_{\mathcal{S}}$ of lookahead traces to compute the memorable values that occur in an output, see Example 7.2, and the list $\mathcal{GLT}_{\mathcal{S}}$ of lookahead traces to determine the memorable values that are used in a guard, see Example 7.3. In addition, a guard lookahead trace in $\mathcal{GLT}_{\mathcal{S}}$ contains a marked parameter to identify the referring parameter in the guard. Let getReferringValue(glt) be the function that returns the value of the referring parameter in a guard lookahead trace glt. Moreover, let getDistinctLookaheadTypes(lt) be the function that returns the set of distinct parameters of type 'l' in a lookahead trace lt.

The procedure for computing memorable values after a given input sequence u is outlined in Algorithms 7.1, 7.2, and 7.3. First, we run all output lookahead traces in $\mathcal{OLT}_{\mathcal{S}}$ after u, see Algorithm 7.1, lines 2–11, to determine the memorable values that occur in future outputs. For the generation of concrete lookahead traces we use Algorithm 7.2, which takes as input a lookahead trace lt and a set of lookahead values L. Set L contains values from u that are possibly interesting in the sense that their recurrence in a concrete lookahead trace might lead to a different future behavior. All permutations of n different lookahead values are computed, where n is the number of distinct parameters of type 'l' in lt, see Algorithm 7.2, lines 2–3 and the **Require** condition. For each permutation a concrete lookahead trace is created by replacing a parameter of type 'f' in lt with a fresh value, a parameter of type 'c' with the value of the according constant, and a parameter of type 'l' with the corresponding value of the permutation, see Algorithm 7.2, lines 5–23. The set of concrete lookahead traces is returned to Algorithm 7.1, where each concrete lookahead trace *colt* is run after u on the SUT, see line 5. The run(u, *colt*) function returns the concrete outputs produced for the *colt* subsequence. Only the values inserted in inputs $i_1...i_n$ are possibly memorable after u, see line 6. However, since we reinsert values from u in *colt* using L, it may be the case that value pm in an output originates from the reinsertion and in fact there is no relation to the value in u. To verify this, we use Algorithm 7.3, which tests whether replacing pm with a fresh value (line 3) leads to the same behavior. If so, then pm is not memorable, see line 5. Algorithm 7.3 only returns the values that are indeed memorable. In line 8 in Algorithm 7.1, we extend this set with values from u that have not been reinserted in *colt*, but occur in an output generated in response to *colt*. These values can only come from u and, thus, are memorable. In line 9 we add the found memorable values to the current node. Second, all combinations of concrete guard lookahead traces are generated and run on the SUT, see Algorithm 7.1, lines 13–15. For each concrete guard lookahead trace, we change the referring value into a fresh value and also execute this trace on the SUT, see again Algorithm 7.3, lines 2–3. If the observable output of the SUT changes, then, according to Case (2) in Definition 7.1, it is relevant for the behavior of the SUT whether the guard is true or not. As a result, we keep r as a memorable value and return it to Algorithm 7.1, where it will be added to $MemV$ of the current node. Once all output and guard lookahead traces have been run, we assume that all memorable values after u have been computed.

Algorithm 7.1 Compute memorable values after u

Input: Two lists of lookahead traces $\mathcal{OLT}_\mathcal{S}$ and $\mathcal{GLT}_\mathcal{S}$, the current node $curN$, and an input sequence $u = i_1...i_n$
Output: A set of memorable values after $u = i_1...i_n$
Function computeMemV($\mathcal{OLT}_\mathcal{S}$, $\mathcal{GLT}_\mathcal{S}$, $curN$, $i_1...i_n$)

1: **if** $curN \neq N_0$ **then**
2: $L :=$ computeMemV($\mathcal{OLT}_\mathcal{S}$, $\mathcal{GLT}_\mathcal{S}$, $curN.parent$, $i_1...i_{n-1}$) $\cup$ values(i_n) ▷ Initialize lookahead values
3: **end if**
4: **for** each output lookahead trace $olt \in \mathcal{OLT}_\mathcal{S}$ **do**
5: $CLT :=$ generateConcreteLookaheadTraces(olt, L) ▷ cf. Algorithm 7.2
6: **for** each concrete output lookahead trace $colt \in CLT$ **do**
7: $ocolt :=$ run(u, $colt$) ▷ Get outputs produced for $colt$
8: $possMemV :=$ values($i_1...i_n$) $\cap$ values($ocolt$) ▷ that also occur in u
9: $memV :=$ verifyMemV($ocolt$, $possMemV \cap L$, u, $colt$) ▷ Algorithm 7.3
10: $memV := memV \cup (possMemV \not\cap L)$ ▷ Definitely memorable
11: addMemV($curN$, $memV$) ▷ Add memorable value(s) to $curN$
12: **end for**
13: **end for**
14: **for** each guard lookahead trace $glt \in \mathcal{GLT}_\mathcal{S}$ **do**
15: $CLT :=$ generateConcreteLookaheadTraces(glt, L) ▷ cf. Algorithm 7.2
16: **for** each concrete guard lookahead trace $cglt \in CLT$ **do**
17: $ocglt :=$ run(u, $cglt$) ▷ Get outputs produced for $cglt$
18: $r :=$ getReferringValue($cglt$) ▷ Get referring value of $cglt$
19: $memV :=$ verifyMemV($ocglt$, $\{r\}$, u, $cglt$) ▷ cf. Algorithm 7.3
20: addMemV($curN$, $memV$) ▷ Add $memV$ to $curN$, possibly empty
21: **end for**
22: **end for**
23: **return** $curN.MemV$ ▷ Return memorable values after u

Algorithm 7.2 Generate concrete lookahead traces

Input: A lookahead trace lt and a set L of lookahead values
Output: A set of concrete lookahead traces
Require: $\mid DL \mid \leq \mid L \mid$, where $DL = \mathsf{getDistinctLookaheadTypes}(lt)$
Function $\mathsf{generateConcreteLookaheadTraces}(lt, L)$

$F :=$ list of fresh values
$n := \mid DL \mid$ ▷ Number of distinct parameters of type 'l' in lt
$P := \{(l_1, l_2, ..., l_n) \in L^n \mid \bigwedge_{1 \leq i < j \leq n} l_i \neq l_j\}$ ▷ Permutations of n distinct lookahead values
$CLT := \{\}$ ▷ Initialize concrete lookahead traces
for each lookahead permutation $perm \in P$ **do**
    $clt :=$ new concrete lookahead trace
    for each lookahead action la in lt **do**
        $cla :=$ new concrete lookahead action with same event primitive as la
        for each parameter lp in la **do**
            $clp :=$ new concrete lookahead parameter
            if lp.type = 'c' **then**
                $clp := lp$.index ▷ Add value of constant
            else if lp.type = 'f' **then**
                $clp := F[lp.\text{index}]$ ▷ Add fresh value
            else
                $clp := perm[lp.\text{index}]$ ▷ Add lookahead value of permutation
            end if
            cla.add(clp)
        end for
        clt.add(ca)
    end for
    $CLT := CLT \cup \{clt\}$ ▷ Add a new concrete lookahead trace
end for
return CLT

Algorithm 7.3 Verify if a set of possible memorable values is indeed memorable

Input: A sequence of concrete outputs os, a set of possible memorable values $possMemV$, a concrete input sequence u, a concrete lookahead trace lt
Output: A set of real memorable values
Function $\mathsf{verifyMemV}(os, possMemV, u, lt)$

for each value pm in $possMemV$ **do**
    $f :=$ first element in $\mathbb{N} \setminus \mathsf{values}(u \cdot lt)$ ▷ Get fresh value
    $os_f := \mathsf{run}(u, lt[f/pm])$ ▷ Replace occurrences of pm with fresh value
    if $os[f/pm]$ equals os_f **then**
        $possMemV$.remove(pm) ▷ pm is NOT memorable
    end if
end for
return $possMemV$ ▷ Return real memorable values

Example 7.5 Consider again the login procedure of Example 1.1. Let

$$\mathcal{OLT}_{\mathcal{S}} = [\mathsf{Register}(('\mathsf{f}',0),('\mathsf{f}',1)),\ \mathsf{Login}(('\mathsf{f}',0),('\mathsf{f}',1)),\ \mathsf{Logout}()] \text{ and}$$

$$\mathcal{GLT}_{\mathcal{S}} = [\mathsf{Login}((\overline{'\mathsf{l}'},0),('\mathsf{l}',1)),\ \mathsf{Login}(('\mathsf{l}',0),(\overline{'\mathsf{l}'},1))],$$

where the horizontal bar denotes the referring parameter. Then the set of concrete output lookahead traces executed after ϵ is {Register(0,1), Login(0,1), Logout()}, where ('f',0) is replaced by fresh value 0 and ('f',1) is replaced by fresh value 1. After ϵ the guard lookahead traces are not executed, because the **Require** condition of Algorithm 7.2 is not fulfilled since the set L of lookahead values is empty. Executing these concrete lookahead traces on the SUT leads to node N_0 depicted in Figure 7.3, which has no memorable values and the newly created nodes N_1, N_2 and N_3. After Register(0,1) the set of concrete output lookahead traces {Register(2,3), Login(2,3), Logout()} is run on the SUT as well as the set of concrete guard lookahead traces {Login(0,1), Login(1,0)}, where $L = \{0, 1\}$ and (0,1) and (1,0) are the permutations of two distinct lookahead values. Moreover, the set of concrete guard lookahead traces is extended with the traces {Login(2,1), Login(2,0)} and {Login(0,2), Login(1,2)}, where the referring value is replaced by a fresh value, and also run on the SUT. Figure 7.3 shows how the observation tree is extended. For presentation purposes we only added a subset of all concrete lookahead traces. Note that after execution of the traces node N_2 has two memorable values {0,1}, see also Example 7.4. □

7.2.3 Lookahead Completeness

Whenever a memorable value has been added to a node, we require an observation tree to be *lookahead complete*: Every memorable value found has to have an origin, i.e., it has to stem from either the memorable values of the parent node or the values in the preceding input.

Definition 7.4 (Lookahead completeness) Let $\mathcal{OT}_{\mathcal{S}} = \langle \mathcal{N}, N_0, \mathcal{E} \rangle$ be an observation tree. If, for all transitions

$$N \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)/\varepsilon_O(d'_1,\ldots,d'_l)} N'$$

of $\mathcal{E}$, we have that $d \in N'.MemV$ implies that either $d \in \{d_1, \ldots, d_k\}$ or $d \in N.MemV$, then $\mathcal{OT}_{\mathcal{S}}$ is called *lookahead complete*.

If a memorable value has been detected, whose origin is unknown, the observation tree is lookahead incomplete. Lookahead completeness ensures that the lookahead traces will find all memorable values $memV(u)$ and that no lookahead of a child node tries to update $memV(u)$ afterwards. Whenever an observation tree becomes lookahead incomplete, the existing lookahead traces are too short to detect all memorable values. For this reason, we need to add a new and longer lookahead trace that finds the missing memorable value(s). Then the entire learning process has to be restarted with the updated lookahead traces to retrieve an observation tree that is lookahead complete.

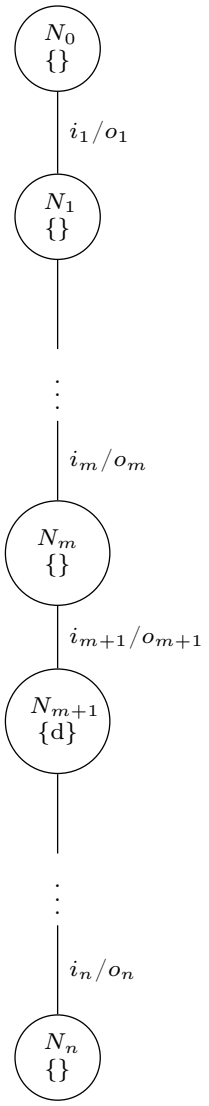


Figure 7.4: *Lookahead incompleteness*

Example 7.6 Let $N_0 \overset{i_1/o_1}{\rightarrow} N_1 \cdots \overset{i_m/o_m}{\rightarrow} N_m \overset{i_{m+1}/o_{m+1}}{\rightarrow} N_{m+1} \cdots \overset{i_n/o_n}{\rightarrow} N_n$ be part of an observation tree and let us try to determine the set of memorable values $memV(u)$ after $u = i_1 \dots i_m\ i_{m+1}$ using $\mathcal{OLT}_\mathcal{S}$, see Figure 7.4. While executing the lookahead traces, let o_n be the output produced in response to input i_n and let o_n contain parameter value d, where d does not occur in inputs $i_{m+1} \dots i_n$. According to Definition 7.1, we add d to the memorable values of node N_{m+1}. Since d is not part of the memorable values of node N_m and it is not contained in input i_{m+1}, we have found a memorable value d for which the observation tree is not lookahead complete, see Definition 7.4. That means the existing output lookahead traces are too short to detect the memorable value. Therefore, we need to add a new and longer lookahead trace to $\mathcal{OLT}_\mathcal{S}$. We create a new lookahead trace for the input string from node N_m to N_n by calling the function createLookaheadTrace($i_{m+1} \dots i_n$, S, *null*) listed in Algorithm 7.4, where S contains the concrete parameter values occurring in $i_1 \dots i_m$ and the referring parameter is not set. The function loops over all concrete parameter values in $i_{m+1} \dots i_n$. If a value is equal to the value of a constant, the lookahead parameter is set to 'c', if it is equal to a value in S, the lookahead parameter is set to 'l', and otherwise it is fresh, so that the lookahead parameter is set to 'f'. Note that only when creating guard lookahead traces the $refParam$ argument is set. The new lookahead trace is added at the end of the $\mathcal{OLT}_\mathcal{S}$ list with the highest index and the entire learning process is restarted with the updated output lookahead traces to retrieve a lookahead-complete observation tree. □

7.2.4 The Behavior of the Lookahead Oracle

Initially, the observation tree only consists of a root node with an empty set of memorable values. Moreover, the list of guard lookahead traces $\mathcal{GLT}_\mathcal{S}$ is empty, because in the beginning there is only the default abstraction, where all values of a parameter are in one large equivalence class, and therefore there are no guards to test. The list of output lookahead traces $\mathcal{OLT}_\mathcal{S}$ is initialized by creating a lookahead trace for every abstract input alphabet symbol using a function similar to Algorithm 7.4. The only differences are that the function takes as single argument an abstract input symbol x and that abstract parameter values are compared instead of concrete values. The initialization ends by executing the computeMemV($\mathcal{OLT}_\mathcal{S}$, $\mathcal{GLT}_\mathcal{S}$, N_0, ϵ) function listed in Algorithm 7.1 to determine the memorable values of the root node.

Algorithm 7.4 Creating a new lookahead trace

Input: A concrete trace $v = i_1 \dots i_n$, a set S of possible memorable values, and the referring parameter (only needed to construct guard lookahead traces)
Output: A new lookahead trace
Function createLookaheadTrace(v, S, $refParam$)

```
 1: lt := new lookahead trace
 2: lookaheadParamCounter := 0
 3: freshValueCounter := 0
 4: val2idx := new map ▷ Maps concrete value to index of lookahead trace param
 5: for each input symbol i in v do
 6:     la := new lookahead action with the same event primitive as i
 7:     for each parameter p in i do
 8:         lp := new lookahead parameter
 9:         if value(p)= γ(c), where c ∈ C then      ▷ Values of p and c are equal
10:             lp := ('c', value(p))
11:         else if value(p) ∈ S then
12:             if val2idx.hasKey(value(p)) then   ▷ If map has entry for value(p)
13:                 lp := ('l', val2idx.get(value(p)))   ▷ Prev. lookahead parameter
14:             else
15:                 lp := ('l', lookaheadParamCounter++) ▷ New lookahead param
16:                 val2idx.put(value(p), lp.index)          ▷ Add new entry to map
17:             end if
18:         else
19:             if val2idx.hasKey(value(p)) then        ▷ If map has entry for value(p)
20:                 lp := ('f', val2idx.get(value(p)))            ▷ Previous fresh value
21:             else
22:                 lp := ('f', freshValueCounter++)                ▷ New fresh value
23:                 val2idx.put(value(p), lp.index)          ▷ Add new entry to map
24:             end if
25:         end if
26:         if p=refParam then    ▷ If p and refParam are the same parameter in v
27:             lp.setReferring()                        ▷ Mark lp as referring parameter
28:         end if
29:         la.add(lp)
30:     end for
31:     lt.add(la)
32: end for
33: return lt
```

The behavior of the lookahead oracle can informally be described as follows:

- Whenever the observation tree is in a current node N and the lookahead oracle receives a concrete input i from the mapper, the oracle selects the edge $N \xrightarrow{i/o} N'$, updates the current node to the new node N', and produces output symbol o. Moreover, the oracle determines the memorable values of the new current node by calling the $\mathsf{computeMemV}(\mathcal{OLT}_{\mathcal{S}}, \mathcal{GLT}_{\mathcal{S}}, N', u)$ function listed in Algorithm 7.1, where u is the sequence of inputs from the root node to N'. The oracle returns the pair (o, ξ) to the mapper, where ξ is a valuation that assigns the memorable values after u, i.e. $memV(u) = \{m_1 \dots m_n\}$, to state variables $v \in V$ of the mapper. The first memorable value m_1 is assigned to state variable v_1, the second memorable value m_2 to state variable v_2, and so on. The remaining state variables are set to $\perp$. More precisely, for all $v \in V$,

$$\xi = v_i \quad = \quad \begin{cases} m_i & \text{if } i \leq n \\ \perp & \text{otherwise.} \end{cases}$$

- Whenever the observation tree is in a node N and the oracle receives a concrete output o from the implementation after sending input i, the observation tree is extended with a new edge $N \xrightarrow{i/o} N'$ and a new child node N' containing an empty set of memorable values.

When selecting the edge $N \xrightarrow{i/o} N'$ in the observation tree, see first item above, this edge already exists in the tree. In the previous step we have determined the output and memorable values for N by running all concrete output and guard lookahead traces. Since the output lookahead traces contain all input alphabet symbols, N' has already been added to the observation tree.

7.3 Mapper

Consider an SUT whose behavior can be described by scalarset Mealy machine $\mathcal{M}_S$ with event signature $\Sigma = \langle T_I, T_O \rangle$. We interact with the SUT via a lookahead oracle that pairs each output o of the SUT with a valuation ξ that assigns to each variable in a given set V either a value that is memorable in the state reached after o, or the undefined value $\perp$. The behavior of the lookahead oracle, as observed by the learner, can be described as a *lookahead extension* of $\mathcal{M}_S$, which is a deterministic Mealy machine $\mathcal{M}_L = \langle I, O \times \mathsf{Val}(V), \mathsf{Runs}(\mathcal{M}), q_0, \rightarrow_l \rangle$, where

1. $\mathsf{Runs}(\mathcal{M})$ is the set of *runs* of $[\![\mathcal{M}_S]\!] = \mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$. A run of a Mealy machine $\mathcal{M}$ is a sequence $\rho = q_0\, i_1\, o_1\, q_1\, i_2\, o_2\, q_2 \cdots i_n\, o_n\, q_n$, for some $n \geq 0$, of alternating states, inputs and outputs of $\mathcal{M}$, beginning with the initial state of $\mathcal{M}$ and such that, for all $1 \leq j \leq n$, $q_{j-1} \xrightarrow{i_j/o_j} q_j$.

2. $\forall \rho, \rho' \in \mathsf{Runs}(\mathcal{M})$, $\rho \xrightarrow{i/(o,\xi)}_l \rho'$ implies $\exists q' : \rho' = \rho\, i\, o\, q' \wedge \forall v \in V : \xi(v) \in memV(q') \cup \{\perp\}$.

In this subsection, we describe the mappers that we use to learn a lookahead extension of $\mathcal{M}_S$. These mappers have three parameters: (a) an event signature $\Sigma = \langle T_I, T_O \rangle$, (b) a finite set V of variables with type $\mathbb{N} \cup \{\bot\}$, (c) a function $F : P \to 2^{V \cup C \cup P}$, where P is the set of parameters that occur in T_I. We require, for all $p, p' \in P$, that if $p' \in F(p)$ then p' occurs before p in the same event primitive. Mapper $\mathcal{A}_{\Sigma,V,F}$ is defined to be the tuple $\langle I \cup O', X \cup Y, R, r_0, \delta, \mathit{abstr} \rangle$, whose elements are described in the next paragraphs.

The concrete actions The set of concrete input symbols is $I = [\![T_I]\!]$ and the set of concrete outputs is $O' = O \times \mathsf{Val}(V)$, where $O = [\![T_O]\!]$.

States and initial state The mapper uses the variables from V to store memorable values. In addition, it has a variable inp with type $I \cup \{\bot\}$, which is used to store the last input action that has occurred. Thus $R = \mathsf{Val}(V \cup \{\mathsf{inp}\})$. The initial state r_0 of the mapper is the valuation in which all variables have value $\bot$.

The update function When an input i occurs, the only effect on the state of the mapper is that inp is updated to i: for all $r \in R$ and $i \in I$, $\delta(r, i) = r[\mathsf{inp} := i]$. When an output $(o, \xi) \in O'$ occurs the variables in V get assigned new values in accordance with ξ, and inp is reset to $\bot$: $\delta(r, o) = \xi \cup \{(\mathsf{inp}, \bot)\}$.

The abstract actions Function $F : P \to 2^{V \cup C \cup P}$ determines the set of abstract inputs. The idea is that in abstract inputs we do not record the actual value of an input parameter p, but only whether or not this value is equal to one of the constants, variables or parameters in $F(p)$.

For each parameter $p \in P$, we introduce an abstract version p^a with type $F(p) \cup \{\bot\}$. The signature T_X of abstract input event primitives is given by

$$T_X \quad = \quad \{\epsilon(p_1^a, \ldots, p_k^a) \mid \epsilon(p_1, \ldots, p_k) \in T_I\}.$$

The set X of abstract input symbols is defined as $X = [\![T_X]\!]$.

Let Q be the set of parameters that occur in T_O. For each parameter $q \in Q$, we introduce an abstract version q^a with type $V \cup C \cup P \cup \{\bot\}$. The signature T_Y of abstract output event primitives is given by

$$T_Y \quad = \quad \{\epsilon(q_1^a, \ldots, q_k^a) \mid \epsilon(q_1, \ldots, q_k) \in T_O\}.$$

The set Y of abstract output symbols is $Y = [\![T_Y]\!] \times (V \to (V \cup C \cup P \cup \{\bot\}))$. The intuition behind the first element of an abstract output is that we do not consider the actual value of an output parameter q, but only whether or not this value is equal to that of one of the variables, constants or parameters of $V \cup C \cup P$. The second element of the abstract output is an update function u that specifies for each variable v the variable or parameter $u(v)$ whose value is assigned to v.

The abstraction function Here we only define the abstraction function for states r that are *injective* in the sense that, for all $v, v' \in V$, $r(v) = r(v') \neq \bot$ implies $v = v'$. We first define the abstraction function *abstr* for input actions.

Suppose r is an (injective) state of the mapper and $i \in I$ is an input. Let $i = \epsilon(d_1, \ldots, d_k)$ and let the corresponding event term be $\epsilon(p_1, \ldots, p_k)$. Then $abstr(r, i) = \epsilon(d_1^a, \ldots, d_k^a)$, where abstract parameter values $d_1^a, \ldots, d_k^a$ are defined as follows. Let $1 \leq i \leq k$. We consider the following four cases in the given order:

1. $d_i = \gamma(c)$, for some constant $c \in F(p_i)$. Then $d_i^a = c$.
2. $d_i = r(v)$, for some variable $v \in F(p_i)$. Then $d_i^a = v$.
3. $d_i = d_j$, for some $j < i$ with $p_j \in F(p_i)$, and for all $l < j$ either $d_l \neq d_i$ or $p_l \notin F(p_i)$. Then $d_i^a = p_j$.
4. Otherwise, $d_i^a = \bot$.

The abstraction function for output actions is defined similarly. Suppose r is an (injective) state of the mapper with $r(\mathsf{inp}) = \epsilon'(e_1, \ldots, e_m)$ and corresponding event term $\epsilon'(p_1, \ldots, p_m)$. Let $d \in \mathbb{N}$. Then $\tau(d) \in V \cup C \cup P \cup \{\bot\}$ is defined by considering the following four cases in the given order:

1. $\tau(d) = c$ if $c \in C$ is a constant with $\gamma(c) = d$.
2. $\tau(d) = v$ if $v \in V$ is a variable with $r(v) = d$.
3. $\tau(d) = p_j$ if $1 \leq j \leq m$ and $d = e_j$, and, for all $l < j$, $e_l \neq d$.
4. $\tau(d) = \bot$.

Suppose $(o, \xi) \in O$ is an output with $o = \epsilon(d_1, \ldots, d_k)$. Then $abstr(r, (o, \xi)) = (\epsilon(\tau(d_1), \ldots, \tau(d_k), u)$, where function $u : V \to (C \cup V \cup P \cup \{\bot\})$ is a symbolic version of the update function defined by: if $\xi(v) = \bot$ then $u(v) = \bot$ else $u(v) = \tau(\xi(v))$.

7.3.1 Concretization

In the previous subsection, we have defined the abstraction function of the mapper only for states that are injective. This restriction is unproblematic during the learning phase: during this phase the mapper component always chooses fresh values for input parameters (except when the abstract action enforces equality to some earlier value), and as a result the mapper component will only reach states that are injective. However, in order to test the validity of an abstract hypothesis we need to extend the definition of the abstraction function, since during the testing phase we may try all possible combinations of input parameter values. As we will see, there are various ways in which we can extend the definition. Consider the fragment of an abstract hypothesis $\mathcal{H}$ displayed in Figure 7.5.

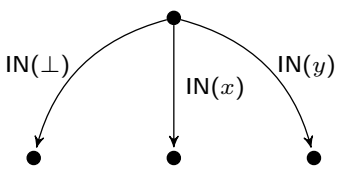


Figure 7.5: *Fragment of abstract hypothesis*

There are two state variables x and y, and a single event primitive $IN(p)$. During learning we have discovered that it is relevant whether the parameter p equals x or y, and thus we have abstract input actions $IN(x)$, $IN(y)$ and $IN(\perp)$. Since the mapper is partially defined, also the concretization of $\mathcal{H}$ is partially defined. Figure 7.6 shows the concretization of the fragment of $\mathcal{H}$ as induced by the partial mapper.

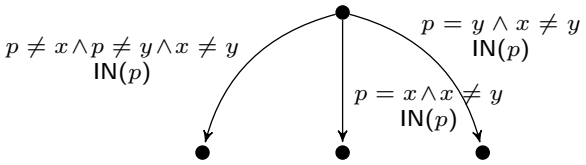


Figure 7.6: *Concretization of abstract hypothesis fragment based on partial mapper*

This Mealy machine is not input enabled and can thus not be used for testing whether hypothesis $\mathcal{H}$ is correct: we need to relax the guards to ensure that transitions are also enabled in the case where $x = y$. The simple solution displayed in Figure 7.7, in which we just remove the guard $x \neq y$, does not work since this Mealy machine is nondeterministic: although we can use it for test generation analysis of counterexamples becomes problematic due to nondeterminism.

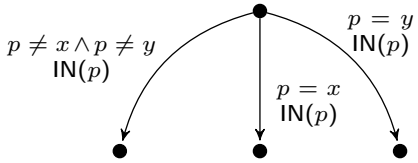


Figure 7.7: *Nondeterministic concretization*

The challenge thus becomes to relax the guards of the various transitions in such a way that all the guards are disjoint, but the resulting Mealy machine is input enabled. In particular, we need to decide which transition to take in the case where certain variables are equal. Figure 7.8 shows a solution in which we take the $IN(\perp)$ transition whenever $x = y$.

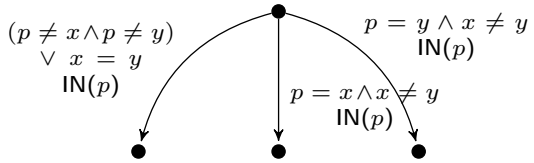


Figure 7.8: *A correct concretization*

In our Tomte tool we have implemented a different solution in which the relaxation of the guards depends on the current state of the abstract hypothesis. The basic idea is that although in general it may be relevant whether a parameter p is equal to variable x or y, this is not necessarily the case for a specific location. In locations in which the value of x is not relevant, the $IN(x)$ and $IN(\perp)$ transitions will generate the same output and have the same target locations. In this case the concretization displayed in Figure 7.9 makes sense.

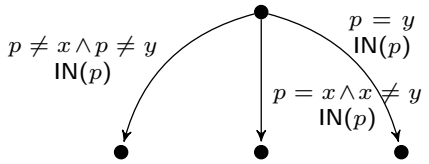


Figure 7.9: *A alternative concretization in case value of x is not relevant*

We can also simplify the concretization by merging the $IN(x)$ and $IN(\bot)$ transitions, as displayed in Figure 7.10.

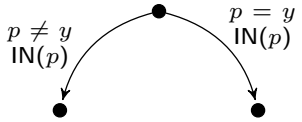


Figure 7.10: *Concretization with merged transitions*

Suppose $\epsilon(p_1, \ldots, p_k)$ is an input event primitive, suppose $1 \leq i \leq k$, and suppose $f \in F(p_i)$ is an abstract parameter value for p_i. Let l be a state of the abstract hypothesis $\mathcal{H}$. Then we say that f is *irrelevant* for p_i in location l if, for each outgoing $\epsilon(d_1, \ldots, d_k)$-transition of l with $d_i = f$, both the output symbol and target state are the same as those of the outgoing $\epsilon(d'_1, \ldots, d'_k)$-transition of l, where $d'_j = \textbf{if } i = j \textbf{ then } \bot \textbf{ else } d_j$.

We define $F_l(p_i)$ to be the set of abstract values that are relevant in location l:

$$F_l(p_i) \quad = \quad \{f \in F(p_i) \mid f \text{ is not irrelevant for } p_i \text{ in } l\}$$

In the concretization that is constructed by Tomte, we assign guard FALSE to all transitions that contain an irrelevant symbol, thus effectively removing these transitions from the concretization. Semantically, we merge these transitions with the transitions in which the irrelevant symbols are replaces by $\bot$. Consider an outgoing $\epsilon(d_1, \ldots, d_k)$-transition of location l in which all the d_i are relevant values. Then in the concretization that is constructed by Tomte we assign the guard $\bigwedge_i g_i$ to this transition, where

$$g_i \quad = \quad \begin{cases} p_i = d_i \wedge PWD(F_l(p_i)) & \text{if } d_i \neq \bot \\ NEQ(p_i, F_l(p_i)) \vee \neg PWD(F_l(p_i)) & \text{if } d_i = \bot \end{cases}$$

Here $NEQ(f, \{f_1, \ldots, f_n\})$ is the predicate stating that f is different from $f_1, \ldots, f_n$:

$$\begin{aligned} NEQ(f, \emptyset) &\equiv \textsf{TRUE} \\ NEQ(f, \{f_1, \ldots, f_n\}) &\equiv NEQ(f, \{f_1, \ldots, f_{n-1}\}) \wedge f \neq f_n \end{aligned}$$

and $PWD(\{f_1, \ldots, f_n\})$ is the predicate that asserts that all f_i are pairwise different:

$$\begin{aligned} PWD(\emptyset) &= \textsf{TRUE} \\ PWD(\{f_1, \ldots, f_n\}) &\equiv PWD(\{f_1, \ldots, f_{n-1}\}) \wedge NEQ(f_n, \{f_1, \ldots, f_{n-1}\}) \end{aligned}$$

7.4 Counterexample-Guided Abstraction Refinement or Lookahead Extension

If the mapper detects that test sequence u is a counterexample, i.e. an observation $(u, s) \in obs_{SUT} - obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$, we need to refine the hypothesis automaton $\mathcal{H}$. The counterexample is used either to

- refine the input abstraction, which leads to an extension of the abstract input alphabet and, accordingly, to (a) new guarded transition(s), or
- extend the set of lookahead traces, which leads to one or more new memorable values and, possibly, to (a) new state variable(s), or
- construct an abstract counterexample to let the *learner* enlarge the size of the hypothesis automaton, which leads to one or more new states.

Before figuring out the source of the counterexample, we first shrink the counterexample since analyzing short traces is usually easier and faster. Shrinking counterexamples or finding minimal counterexamples is a known technique in model-based testing and model checking [97, 18]. Koopman et al. [97], for instance, apply different heuristics like binary search and eliminating single transitions, larger chunks of inputs and cycles. Their measurements show that a combination of several heuristics performs best, but that cycle elimination alone is effective, cheap to execute and scales very well. In a similar way, we remove cycles in the counterexample using the hypothesis model $\mathcal{H}$. Let u^a be the abstract counterexample input sequence obtained via mapper $\mathcal{A}$ and $\tau_{\mathcal{A}}(u, s)$. If a run of u^a on $\mathcal{H}$ contains cycles, there is a fair chance that the SUT contains the same cycles. Our counterexample reduction algorithm sorts the cycles from large to small and tries to remove them in that order.
Once all cycles have been removed, we process the reduced counterexample using Algorithms 7.5 and 7.6 to decide how to refine the hypothesis automaton $\mathcal{H}$.

Algorithm 7.5 Process counterexample

Input: Concrete reduced counterexample (u, s)
Output: Abstract counterexample (u^a, s^a) for *learner*, a new entry v for $F(p)$ in abstraction table, or an extended list of guard lookahead traces $\mathcal{GLT}_{\mathcal{S}}$

```
1: (u^a, s^a) := τ_A(u, s)                          ▷ Abstract version of counterexample
2: s^a_f := runFresh(u^a)   ▷ Fresh trace on SUT via lookahead oracle (Section 7.1)
3: if (u^a, s^a_f) ∉ obs_H then
4:     return (u^a, s^a_f)                  ▷ Return abstract counterexample to learner
5: else
6:     return refineAbstractionOrExtendLookahead(u, s)        ▷ cf. Algorithm 7.6
7: end if
```

Similar to Section 6.2 we consider two different cases: 1) the case that the *learner* has not found all states and we need to forward the abstract counterexample to the *learner* to solve the problem and 2) the case that our current abstraction is too coarse and we need to refine it by 2a) a new input abstraction

or 2b) an extended lookahead if no input abstraction can be defined. The procedure to decide which of the two cases we are facing is still the same, see Algorithm 7.5: We convert the input sequence of the counterexample into a fresh trace by replacing all duplicate values that are not covered by existing input abstractions into fresh values. In this way we assure that we have constructed a trace for which no further abstraction refinement is possible. Hence, if it turns out that the fresh trace is still a counterexample, we know that only the *learner* can resolve it. We execute the fresh trace on the SUT via the lookahead oracle to determine the abstract output sequence s_f^a using the memorable values computed in the tree, see $\mathsf{runFresh}(u^a)$ function in line 2. If the hypothesis does not contain the new observation (line 3), then hypothesis $\mathcal{H}$ is incorrect and the abstract counterexample (u^a, s_f^a) is forwarded to the *learner*, see line 4. Otherwise, the problem can be solved with a new input abstraction or lookahead trace. In this case the abstract outputs in s_f^a differ from s^a due to less equalities between data values in the concretized input of the fresh trace, which lead to a different path through the SUT and, thus, to a different outcome.

The procedure for counterexample-guided abstraction refinement or lookahead extension (CEGAROLE) is shown in Algorithm 7.6. Similar to Algorithm 6.1, we loop over all black values, which refer to possible abstractions, until we have found either a new input abstraction or a new lookahead trace.

1. **Adding new guarded transition(s):** We change every black value b into a fresh value f and check if the observable output of the SUT changes, see line 4. If this is the case and, moreover, b is equal to the value of constant c, then we obtain a new entry for the abstraction table, see $(\mathsf{param}(b), c)$ in line 6. Also if changing the previous occurrence of b, i.e. $\mathsf{source}(b)$, into a fresh value leads to a change of the observable output (line 9), we have found a relation between two data values, whose equality is relevant for the behavior of the SUT. In order to add a new entry to the abstraction table, we need to define the relation between $\mathsf{param}(b)$ and $\mathsf{param}(\mathsf{source}(b))$ in the guard. If b is equal to the value of some state variable v_j (line 10), the relation $(\mathsf{param}(b), v_j)$ is the new entry for the abstraction table, see line 11. If $\mathsf{param}(b)$ and $\mathsf{param}(\mathsf{source}(b))$ are parameters in the same action (line 12), where $\mathsf{param}(\mathsf{source}(b))$ has a lower index than $\mathsf{param}(b)$, then the relation $(\mathsf{param}(b), \mathsf{param}(\mathsf{source}(b)))$ is added as a new entry, see line 13.

2. **Adding new state variable(s):** However, if we cannot find a match (line 14-21), then $\mathsf{param}(b)$ is equal to a state variable, which has not been detected so far or that has been reset to $\perp$, i.e. a memorable value has not been remembered long enough. To identify that $\mathsf{source}(b)$ is memorable and has to be stored in a state variable, we try to add at least one new lookahead trace to $\mathcal{GLT}_\mathcal{S}$. Let $i_k = \mathsf{action}(\mathsf{param}(\mathsf{source}(b)))$ and $i_m = \mathsf{action}(\mathsf{param}(b))$ be the inputs that contain the first and second occurrence, respectively, of the value whose equality is relevant for the behavior of the SUT. Let the reduced counterexample trace be $q_0 \overset{i_1/o_1}{\to} q_1 \cdots \overset{i_k/o_k}{\to} q_k \cdots \overset{i_m/o_m}{\to} q_m \cdots \overset{i_n/o_n}{\to} q_n$. Then, the list of guard lookahead traces is extended as follows: If the lookahead trace for $i_{k+1} \dots i_n$ is not an element of $\mathcal{GLT}_\mathcal{S}$ (lines 15 – 17), we add it, together with the suffix-closed set of lookahead traces from $i_{k+2}, \dots, i_n$

to $i_m, \ldots, i_n$, at the end of the $\mathcal{GLT}_\mathcal{S}$ list, see line 18. Here, suffix-closedness means that if $i_{k+1}, \ldots, i_n$ is a new lookahead trace, then also $i_{k+j}, \ldots, i_n$ is a new lookahead trace for arbitrary $1 \leq j \leq (m-k)$. If no new lookahead trace can be found, no new information can be deduced from black value b to solve the counterexample. Therefore, we pick the next occurrence of a black value b from u and analyze it, starting at Item 1.

Algorithm 7.6 Abstraction refinement or lookahead extension

Input: Concrete reduced counterexample $(u = i_1 \ldots i_n, s)$
Output: New input abstraction (p, v) with v new entry for $F(p)$ in abstraction table, or an extended list of guard lookahead traces $\mathcal{GLT}_\mathcal{S}$
Function refineAbstractionOrExtendLookahead(u, s)

```
1: while abstraction not found or GLT_S not extended do
2:     pick first occurrence of a black value b from u
3:     u' := u, where param(b) is set to a fresh value f
4:     if output s'[f/b] of u' on SUT ≠ s then
5:         if b = γ(c), where c ∈ C then        ▷ If b is equal to value of constant c
6:             return (param(b), c)                    ▷ New abstraction to constant
7:         else
8:             u'' := u, where param(source(b)) is set to a fresh value f
9:             if output s''[f/b] of u'' on SUT ≠ s then
10:                if b = r(v_j), where v_j ∈ V then     ▷ If b is equal to value of v_j
11:                    return (param(b), v_j)     ▷ New abstraction to state variable
12:                else if action(param(b)) = action(param(source(b))) then
13:                    return (param(b), param(source(b)))  ▷ New abstr. to param
14:                else
15:                    i_k := action(param(source(b)))
16:                    lt := createLookaheadTrace(i_{k+1} ... i_n, values(i_1 ... i_j),
   param(b))                                                    ▷ cf. Algorithm 7.4
17:                    if lt ∉ GLT_S then      ▷ If new guard lookahead trace found
18:                        GLT_S := GLT_S ∪ {lt} ∪ suffixClosedSubtraces(lt)
19:                        return GLT_S                  ▷ Return extended GLT_S list
20:                    end if
21:                end if
22:            else
23:                u := u''                ▷ Counterexample with fewer black values
24:            end if
25:        end if
26:    else
27:        u := u'                         ▷ Counterexample with fewer black values
28:    end if
29: end while
```

Finally, Algorithm 7.6 will succeed either in finding a new input abstraction, which is added as an entry in the abstraction table, or in extending the list of guard lookahead traces. The *learner* is restarted with the updated information. If

a new input abstraction has been added, an output lookahead trace for the new abstract input alphabet symbol is created using Algorithm 7.4. Thus, the next hypothesis will be constructed using an enlarged observation tree.

7.5 The Resulting Algorithm

The overall learning algorithm is presented in Figure 7.11. Rounded rectangles represent actions, diamonds represent decisions, rectangles on the left represent inputs to the learning algorithm and rectangles on the right represent the outputs. Wherever appropriate, the algorithm corresponding to an action or decision is given.

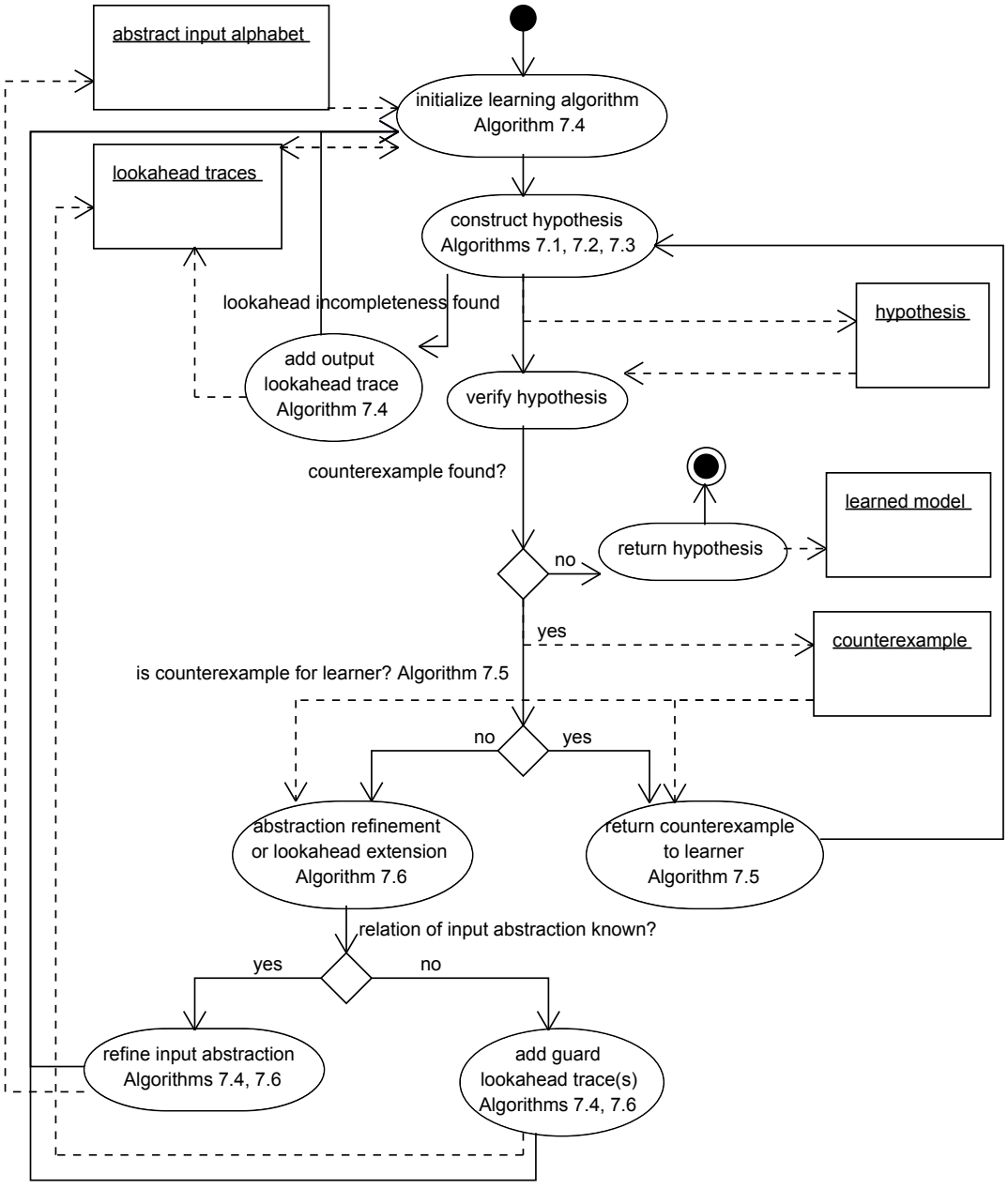


Figure 7.11: *Overall learning algorithm*

The learning process is performed in two alternating phases: the construction of a hypothesis and the verification of the hypothesized model, which is again refined if a counterexample has been found. The cycle is only interrupted when

- no counterexample has been found, which means the hypothesis is correct according to the MBT tool, and the learning has succeeded or
- the inputs to the learning algorithm, i.e. abstract input alphabet or lookahead traces, need to be refined such that the entire learning process has to be restarted.

7.6 Example Application

We have implemented the algorithms presented in this chapter in the Tomte tool. In this section, we illustrate how our algorithms work together by means of a simple system, which we successfully learned with the Tomte tool. In contrast to the examples in Section 6.3, input abstractions are expressed in a more general way and the flow of the overall algorithm has changed due to keeping track of lookahead traces.

As example application we use a FIFO-set with a capacity of two, similar to the one presented in [80]. A FIFO-set corresponds to a queue in which only different values can be stored, see Figure 7.12. As in the stack of 7.1, there are a $\mathsf{Push}(p)$ input that tries to insert a value in the set and a $\mathsf{Pop}()$ input that tries to retrieve a value from the set. The output in response to a $\mathsf{Push}(p)$ input is OK if p could be added successfully or NOK if p is already an element of the set or if the set is full. The output in response to a $\mathsf{Pop}()$ input is $\mathsf{Out}(x)$, where x is the first value that has been added to the set and not has been returned, or NOK if the set is empty.

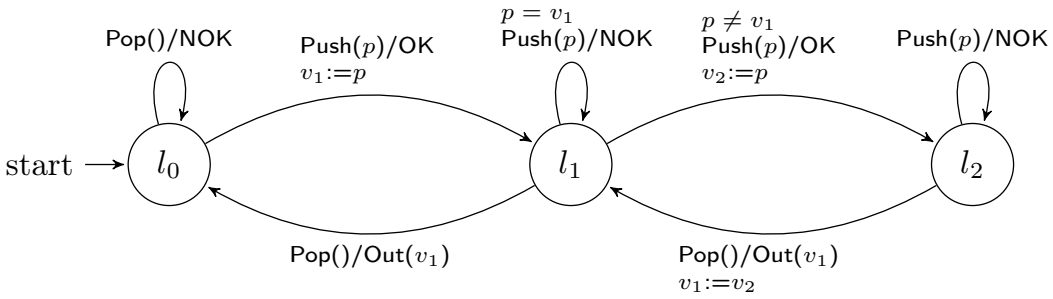


Figure 7.12: *FIFO-set with a capacity of 2 modeled as a scalarset Mealy machine*

The learning is performed as follows. The *learner* is initialized with the abstract input symbols $\mathsf{Push}(\bot)$ and $\mathsf{Pop}()$. The lookahead oracle is equipped with the list of output lookahead traces $\mathcal{OLT}_{\mathcal{S}} = [\mathsf{Push}(('\mathsf{f}',0)), \mathsf{Pop}()]$, an empty list of guard lookahead traces $\mathcal{GLT}_{\mathcal{S}} = []$, and an observation tree with a root node for which the memorable values $\{\}$ have been determined by executing the concrete output lookahead traces $\{\mathsf{Push}(0), \mathsf{Pop}()\}$. The *learner* starts by asking output queries using the two abstract input symbols. The mapper concretizes these symbols as usual by selecting the smallest fresh value for every abstract parameter value $\bot$.

All concrete output queries are forwarded from the mapper to the SUT and, in addition, are stored in the observation tree. Moreover, the memorable values are computed after every output query by running the concrete lookahead traces on the SUT and storing them in the tree. While learning the first hypothesis of the FIFO-set the observation tree depicted in Figure 7.13 is constructed.

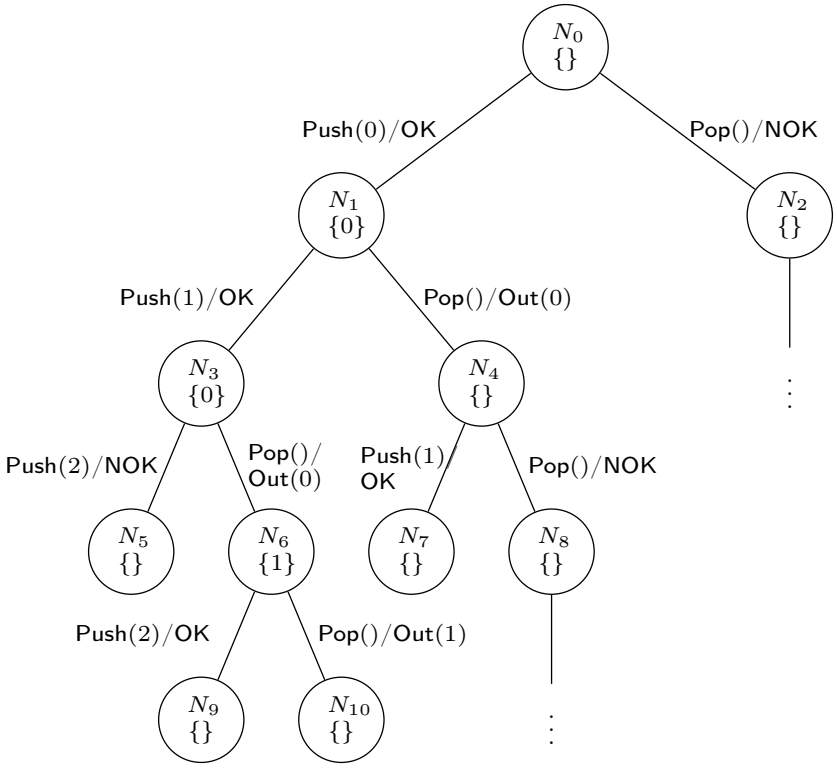


Figure 7.13: *Observation tree of FIFO-set*

This observation tree is no longer lookahead complete since memorable value 1 of node N_6 is not part of the memorable values of node N_3 and has not been inserted via the previous Pop() input. Therefore, we extend $\mathcal{OLTS}$ with the output lookahead trace Pop() Pop() and restart the entire learning process with the updated lookahead traces to retrieve a lookahead-complete observation tree. Now, the *learner* succeeds to construct the abstract hypothesis shown in Figure 7.14.

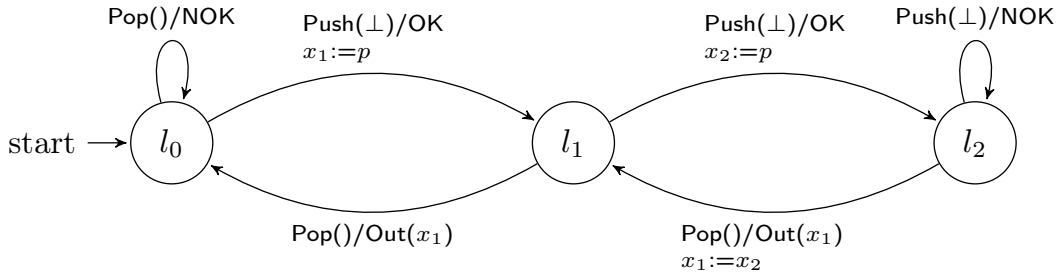


Figure 7.14: *First hypothesis of the FIFO-set*

This hypothesis does not check if the same value is inserted twice since the mapper only uses fresh values in the output queries. During hypothesis verification the mapper selects random values from a small range for every abstract parameter

value $\perp$. In this way it will find a concrete counterexample input trace, e.g. Pop() Push(9) Pop() Push(3) Push(3), see line 2 below, for which the SUT produces a NOK output and the hypothesis generates an OK, see lines 3 and 4.

1:	abstract inputs:	Pop()	Push($\perp$)	Pop()	Push($\perp$)	Push($\perp$)
2:	concrete inputs:	Pop()	Push(9)	Pop()	Push(3)	Push(3)
3:	outputs SUT:	NOK	OK	Out(9)	OK	NOK
4:	outputs hypothesis:	NOK	OK	Out(9)	OK	OK

First, we shrink the counterexample as discussed in Section 7.4. The longest cycle of running the abstract inputs in line 1 on the abstract hypothesis in Figure 7.14 is from l_0 to l_0 with Pop() Push($\perp$) Pop() and has length three. Removing this cycle from the concrete counterexample inputs results in Push(3) Push(3), see line 6 below, for which the SUT and hypothesis still produce different outputs, see lines 7 and 8. To determine if it is a counterexample for the *learner*, see Algorithm 7.5, we convert the reduced concrete counterexample inputs into a fresh trace (line 9) and run it on the SUT via the lookahead oracle. The concrete outputs returned by the SUT are OK OK, see line 10. Since, after abstraction, the outputs of the fresh trace are also produced by the abstract hypothesis (it also generates two successive OK outputs), we need to refine the input abstraction according to Algorithm 7.6.

5:	abstract inputs:	Push($\perp$)	Push($\perp$)
6:	concrete inputs:	Push(3)	Push(3)
7:	outputs SUT:	OK	NOK
8:	outputs hypothesis:	OK	OK
9:	fresh trace:	Push(1)	Push(2)
10:	outputs fresh trace:	OK	OK

There is only one black value in the counterexample, see line 12 below, which we replace with a fresh value, e.g. with value 7, see line 13. Running the new trace on the SUT gives a different output than before (line 14 vs. line 7). Since there are no constants defined in the FIFO-set, we try the same with the source of the black value. We reset the black value and replace the first 3 with a fresh value, e.g. 2, see line 15. Running the new trace on the SUT gives again a different output than produced by the original counterexample (line 16).

11:	abstract inputs:	Push($\perp$)	Push($\perp$)
12:	**black** and red values:	Push(3)	Push(**3**)
13:	**black value** $\rightarrow$ fresh value:	Push(3)	Push(7)
14:	outputs SUT:	OK	OK
15:	source value $\rightarrow$ fresh value:	Push(2)	Push(**3**)
16:	outputs SUT:	OK	OK

Thus, we have found a relation between two data values, whose equality is relevant for the behavior of the SUT. In order to add a new entry to the abstraction table, we need to know, where the 3 comes from. As mentioned before, there are no

constants defined. Moreover, there are no other parameters in the $\mathsf{Push}(p)$ input with a lower index. Therefore, the only possible origin is a state variable. When handling the second Push input of the original counterexample, which contains the black value, there is only one state variable set, i.e. $x_1 = 3$, see Figure 7.14. Thus, we have found the origin of the new abstraction and add it to the abstraction table, see Table 7.1.

Parameter	**Abstraction(s)**
Push p	x_1

Table 7.1: *Abstraction table for FIFO-set*

The alphabet of the *learner* is extended with a new input symbol $\mathsf{Push}(x_1)$ and a corresponding output lookahead trace is added to $\mathcal{OLT}_{\mathcal{S}}$, which now contains four elements: $\mathcal{OLT}_{\mathcal{S}} = [\mathsf{Push}((\mathsf{'f'},0)), \mathsf{Pop}(), \mathsf{Pop}()\ \mathsf{Pop}(), \mathsf{Push}((\mathsf{'l'},0))]$. Again, the entire learning process is restarted from scratch. The next hypothesis learned is similar to the model in Figure 7.12 and the learning algorithm stops.

7.7 Experimental Results

As mentioned before, we have implemented the algorithms presented in this chapter in the Tomte tool. We have connected our Tomte tool to the LearnLib library [147, 114] for regular inference and have performed several experiments to demonstrate the applicability of our approach. As MBT tool we have implemented a random test suite that by default is configured with 1000 test traces of length 100. Using CADP, we verified that the learned models indeed are correct, i.e., equivalent to the SUT that has been generated from the extended finite state machine modeled in Uppaal [24]. We have run each experiment ten times with different seeds. For every experiment we have measured the following data and determined the average over the successful runs together with the standard deviation:

- **succ**: number of runs succeeded of total number of runs
- **states**: total number of states of the learned abstract model
- **abs res**: total number of output query resets received from LearnLib
- **abs inp**: total number of output queries received from LearnLib (abstract input symbols)
- **learn res**: total number of resets sent to SUT during learning
- **learn inp**: total number of concrete input symbols sent to SUT during learning
- **test res**: total number of resets sent to SUT during testing
- **test inp**: total number of concrete input symbols sent to SUT during testing (without last test, where no counterexample has been found)

- **ana res**: total number of resets sent to SUT during counterexample analysis
- **ana inp**: total number of concrete input symbols sent to SUT during counterexample analysis
- **abs CE**: total number of abstract counterexamples sent to LearnLib
- **abs ref**: total number of input abstractions refined
- **GLT**: total number of guard lookahead traces added (without suffix-closed subtraces)
- **OLT**: total number of lookahead incompleteness detected

We also measured the time of our experiments, but we do not mention these numbers in our learning statistics as in our opinion the other measures are more relevant. However, for every series of experiments, we give an overall impression of the time needed for learning, testing, and counterexample analysis.

First, we have employed our tool to repeat the experiments of the previous chapter. The learning results are presented in Table 7.2. Comparing the new results to the results of the previous chapter shows that for most systems the number of states in the learned model is still the same and that the number of sequences of output queries received from LearnLib (**Learning queries** in Table 6.2 versus **abs res** in Table 7.2) just has changed slightly. Only the values for the session initiation protocol and login procedure have changed considerably. For the session initiation protocol, the increase in output queries is due to two more guards that have been inferred, compare **Input refinements** in Table 6.2 to **abs ref** in Table 7.2. Each of these guards is tested in every state of the hypothesized model to make it input-enabled. For the login procedure, the number of output queries has increased due to more learning rounds that are required in the new setting. In the previous version of the Tomte tool learning is only restarted from scratch whenever the abstraction is refined. In our new algorithm, it is also restarted whenever we add (a) new guard lookahead trace(s), see Figure 7.11. Instead of 3 learning rounds, now we need 7 rounds, including 4 restarts because of new guard lookahead traces. As a result, the same abstract output queries are asked and counted several times. However, from a performance perspective, this is not a problem since we use the observation tree as cache.

In comparison to the previous experiments, we now derive memorable values automatically, which goes along with additional costs. For the identification of memorable values not only the output queries received from the *learner* are concretized and forwarded to the SUT, but also a number of concrete lookahead traces. When looking at the ratio of output queries received from LearnLib (**abs inp**) to the number of concrete inputs really sent to the SUT (**learn inp**), there is quite some difference in the costs for the various systems. In Table 7.2, the ratio of abstract output queries to concrete inputs ranges from 1:3 (alternating bit protocol receiver) to 1:9 (login procedure). For the other experiments performed in this section, the ratio lies between 1:2 and 1:5. Although more concrete inputs have been sent to the SUT, times for learning, testing, and counterexample analysis for the SUTs in Table 7.2 are rather low, not exceeding 2 seconds.

	succ	states	abs res	abs inp	learn res	learn inp	test res	test inp	ana res	ana inp	abs CE	abs ref	GLT	OLT
Alternating bit protocol sender														
avg	10/10	7	185	698	465	2459	2	93	9	19	0	1	0	0
stddev		0	0	0	0	2	2	181	5	16	0	0	0	0
Alternating bit protocol receiver														
avg	10/10	4	134	382	271	1168	4	145	20	58	0	2	0	0
stddev		0	0	0	0	0	1	120	3	10	0	0	0	0
Alternating bit protocol channel														
avg	10/10	2	28	57	67	210	0	0	0	0	0	0	0	0
stddev		0	0	0	0	0	0	0	0	0	0	0	0	0
Biometric passport														
avg	10/10	5	2167	7708	8769	43373	179	17551	53	265	0	3	0	0
stddev		0	40	234	5	35	128	12859	6	33	0	0	0	0
Session initiation protocol														
avg	10/10	10	2564	9435	7491	50263	143	13024	387	2784	2	4	2	0
stddev		0	140	576	1382	11531	64	6382	112	884	1	0	0	0
Login procedure														
avg	10/10	3	703	2137	3770	19586	83	7286	56	240	0	2	4	0
stddev		0	0	0	1	0	64	6396	19	78	0	0	0	0
River crossing puzzle														
avg	10/10	9	514	2554	2114	14452	27	1730	96	579	2	4	0	0
stddev		0	43	184	88	813	12	1227	14	178	0	0	0	0
Palindrome/repnumber checker														
avg	10/10	1	1713	3181	8367	24717	241	22417	72	122	0	9	0	0
stddev		0	81	156	3	6	177	17653	6	11	0	0	0	0

Table 7.2: *Learning statistics for the SUTs of the previous chapter, see Sections 6.3 and 6.4*

In a second series of experiments, we have applied our Tomte tool to learn models of the bounded retransmission protocol (BRP) [75, 55]. A detailed description of the protocol and six faulty mutant implementations we made can be found in Section 8.3 and Appendix 8.A of the next chapter. The learning statistics are shown in Table 7.3. Note that mutant 4 and 5 generate the same results as the reference implementation, because they only differ in abstract parameter values produced: While mutant 4 returns an $\mathsf{OFRAME}(c_1, c_0, c_0, v_1)$ output from state SF to WA, where $n == 2$, the reference implementation generates $\mathsf{OFRAME}(c_0, c_1, c_0, v_1)$. In a similar way, mutant 5 produces a $\mathsf{OCONF}(c_0)$ output from state SC to INIT, where $n == 2$, while the reference implementation returns $\mathsf{OCONF}(c_2)$. Inferring mutant 6 requires the most output queries (**abs inp**) and concrete inputs sent to the SUT (**learn inp**). This is due to a guard that has to be inferred, see the additional OFRAME transition from state SF to WA in Figure 8.9 in Appendix 8.A. Therefore it is not surprising that learning mutant 6 takes the longest with 2 seconds for learning and 1 second for testing and counterexample analysis.

In a third series of experiments, we have applied our tool to learn models of different data structures. Table 7.4 presents the learning results for a queue and a stack as illustrated in Figures 7.15 and 7.1. We have incrementally scaled up the capacity of the queue and stack from 2 to 40 elements to test the limits of our tool. Remarkably, both data structures produce exactly the same learning statistics for the same capacity so that we have merged them in one table. Since the standard deviation was 0 for all measures, we have removed the corresponding rows from the table. Increasing the number of elements in the queue or stack leads to a (approximately) cubic growth of total output queries, see **abs inp** in both tables. Also the total number of concrete inputs sent to the SUT to construct the observation tree and to derive memorable values grows (approximately) cubically, see **learn inp**. However, it is particularly striking that the total number of test inputs (**test inp**) is always 0 and independent of the size of the data structure. The reason for this is that during construction of the first hypothesis of a queue or stack with capacity n the observation tree detects $n-1$ times lookahead incompleteness, see **OLT** in Table 7.4. Every time the Pop() output lookahead trace is extended with another Pop() symbol. Using the longer lookahead trace, a new memorable value and a new state are found without any additional effort. While exploring the near future of the new state again lookahead incompleteness is detected and again the output lookahead trace is extended with another Pop() symbol, and so on. When LearnLib finally succeeds to construct the first hypothesis, the model contains all $n + 1$ states and all n state variables so that our MBT tool cannot find a counterexample. Thus, the time needed for testing and counterexample analysis is 0 seconds, because we did not measure the last test and there was no counterexample to analyze. The time needed for learning did not exceed 7 seconds, even for a stack or queue with a capacity of 40.

In another series of experiments, we have applied our Tomte tool to learn a 2-dimensional stack of overall capacity 4, see Appendix 7.A.2. We did not scale up the capacity of the 2-dimensional stack. The learning results are presented in Table 7.5. The models could be inferred rather quickly with our tool, not exceeding 1 second for learning, testing, or counterexample analysis.

	succ	states	abs res	abs inp	learn res	learn inp	test res	test inp	ana res	ana inp	abs CE	abs ref	GLT	OLT
BRP reference implementation, mutant 4, and mutant 5														
avg	10/10	38	699	6013	2482	26131	11	971	259	2960	2	0	0	1
stddev		0	0	0	0	29	6	601	35	643	0	0	0	0
BRP mutant 1														
avg	10/10	38	584	4415	2177	20810	12	1144	66	467	1	0	0	1
stddev		0	0	0	2	30	9	869	16	173	0	0	0	0
BRP mutant 2														
avg	10/10	32	591	4674	1836	17749	6	390	221	2371	2	0	0	1
stddev		0	0	0	0	19	3	255	35	473	0	0	0	0
BRP mutant 3														
avg	10/10	38	768	6004	2471	23726	16	1380	208	1842	3	0	0	1
stddev		0	56	522	7	63	11	1053	54	594	0	0	0	0
BRP mutant 6														
avg	10/10	60	2454	21426	9094	95462	29	2403	412	4160	3	1	0	1
stddev		0	293	2758	93	1018	13	1358	168	2044	1	0	0	0

Table 7.3: *Learning statistics for the BRP implementation and mutants 1-6*

	succ	states	abs res	abs inp	learn res	learn inp	test res	test inp	ana res	ana inp	abs CE	abs ref	GLT	OLT
Queue(2) and Stack(2)														
avg	10/10	3	33	70	39	148	0	0	0	0	0	0	0	1
Queue(5) and Stack(5)														
avg	10/10	6	123	370	111	742	0	0	0	0	0	0	0	4
Queue(10) and Stack(10)														
avg	10/10	11	393	1810	311	3612	0	0	0	0	0	0	0	9
Queue(15) and Stack(15)														
avg	10/10	16	813	5075	611	10132	0	0	0	0	0	0	0	14
Queue(20) and Stack(20)														
avg	10/10	21	1383	10915	1011	21802	0	0	0	0	0	0	0	19
Queue(25) and Stack(25)														
avg	10/10	26	2103	20080	1511	40122	0	0	0	0	0	0	0	24
Queue(30) and Stack(30)														
avg	10/10	31	2973	33320	2111	66592	0	0	0	0	0	0	0	29
Queue(35) and Stack(35)														
avg	10/10	36	3993	51385	2811	102712	0	0	0	0	0	0	0	34
Queue(40) and Stack(40)														
avg	10/10	41	5163	75025	3611	149982	0	0	0	0	0	0	0	39

Table 7.4: *Learning statistics for a queue and a stack with capacity 2 to 40*

	succ	states	abs res	abs inp	learn res	learn inp	test res	test inp	ana res	ana inp	abs CE	abs ref	GLT	OLT
2-dimensional stack														
avg	10/10	13	463	1844	1333	8956	0	0	0	0	0	0	0	3

Table 7.5: *Learning statistics for a 2-dimensional stack of overall capacity 4*

In a last series of experiments, we have increased the complexity of our queue by adding a guard to every $\mathsf{Push}(p)$ transition that checks whether p already is an element of the data structure. The resulting scalarset Mealy machine is the FIFO-set depicted in Figure 7.12, which has been discussed in the previous section. Again we have gradually scaled up the capacity of the FIFO-set to test the limits of our tool. Using the test setup of the previous experiments (1000 test traces of length 100), we quickly reach the boundaries of our tool. Already for a FIFO-set with 14 elements it is not possible to infer all 10 runs successfully. On account of this, we investigated the effects of increasing the length of the test traces. The results for 1000 test traces of length 1000 are shown in Table 7.6. Here, it is possible to successfully learn the FIFO-set with up to 25 elements. The reason for this is quite obvious. For the FIFO-set guards have to be inferred. In contrast to the derivation of memorable values, which we retrieve for free for these data structures, inferring guards is expensive. Guards can only be deduced from counterexamples, which may be hard to find. The larger the capacity of the FIFO-set, the more difficult it is to find a counterexample with random testing for the guards deep in the data structure. Thus, the bottleneck does not lie in our tool, but in the MBT tool. If we would increase the length of the test traces even more or would improve the testing algorithm, most likely we would be able to infer FIFO-sets with larger capacities. Due to the higher complexity of the data structure, also the times have increased: For a FIFO-set with 35 elements the total learning time was 12:09 minutes, the total testing time 13:38 minutes, and the total time for counterexample analysis was 7 seconds.

Our Tomte tool and all models can be downloaded at `http://www.tomte.cs.ru.nl/`.

7.8 Comparison of Different Approaches

As already mentioned in the related work section of the introduction, the LearnLib tool [114] and our Tomte tool implement quite similar algorithms for fully automatically inferring large or infinite-state systems. Therefore, it is worthwhile to examine the differences between both tools in more detail. LearnLib implements the approach to inferring register automata and register Mealy machines as presented in [81, 80]. Register Mealy machines are almost identical to scalarset Mealy machines, which can be learned with the Tomte tool. Our algorithms are motivated by previous cases studies, where a mapper component has been constructed manually. Automatically generating a mapper for any type of SUT is a mammoth task since the SUT may contain any type of operation, which is hard to infer in a black-box setting. Therefore, we decided to focus on a restricted class of systems, which also comprises the SIP and biometric passport case study. Both classes of systems have been developed independently of each other, but by mutual agreement a standardized XML format has been introduced, which is supported by both tools. This did not affect the framework or inner workings of the tools. They still reveal a number of differences, which will be addressed in the remainder of this section.

	succ	states	abs res	abs inp	learn res	learn inp	test res	test inp	ana res	ana inp	abs CE	abs ref	GLT	OLT
FIFO-set(2)														
avg	10/10	3	73	172	99	423	1	20	6	17	0	1	0	1
stddev		0	0	0	0	2	0	25	1	5	0	0	0	0
FIFO-set(5)														
avg	10/10	6	769	3088	1104	8443	9	2158	80	705	0	4	0	4
stddev		0	0	0	5	31	1	1098	40	507	0	0	0	0
FIFO-set(10)														
avg	10/10	11	6669	44608	10760	139708	23	7526	446	7029	0	9	0	9
stddev		0	0	0	22	277	8	7317	210	4918	0	0	0	0
FIFO-set(15)														
avg	10/10	16	26944	249928	45241	827202	46	23005	1431	33652	0	14	0	14
stddev		0	0	0	25	439	11	11389	211	9657	0	0	0	0
FIFO-set(20)														
avg	10/10	21	75844	896923	129628	3056149	94	63422	4169	106467	0	19	0	19
stddev		0	0	0	54	1138	31	30834	565	14495	0	0	0	0
FIFO-set(25)														
avg	10/10	26	172619	2478468	297599	8590181	362	325722	8755	261624	0	24	0	24
stddev		0	0	0	56	1356	205	205325	876	29506	0	0	0	0
FIFO-set(30)														
avg	6/10	31	341519	5764938	591668	20206862	761	718060	15714	620479	0	29	0	29
stddev		0	0	0	72	2112	319	319098	1427	232984	0	0	0	0
FIFO-set(35)														
avg	1/10	36	611794	11866708	1062979	41924038	1907	1858627	24842	1069710	0	34	0	34
stddev		0	0	0	0	0	0	0	0	0	0	0	0	0

Table 7.6: *Learning statistics for a FIFO-set with capacity 2 to 35*

7.8.1 Modularity

Both tools differ in the general architecture of their learning framework as discussed below.

Independent components In Tomte, the structure is more modular by splitting different functions into separate components, e.g. the mapper takes care of abstraction and concretization and the lookahead oracle is responsible for the derivation of memorable values. The *learner* component has not been changed, meaning that it still functions on the abstract level. Hence, the basic learning algorithm is independent of the data and, as a result, can be exchanged with any other learning algorithm. Moreover, Tomte does not need to take care of *learner* related tasks like identifying states. In contrast, LearnLib functions on the concrete level by extending the *learner* component with data. The underlying data structure of the learning algorithm is adapted to store all (relevant) data values. Similar to the lookahead oracle in Tomte, it stores in every cell of the observation table a concrete tree that comprises the future behavior of the corresponding state. Since in Tomte there is no notion of states in the lookahead oracle, the future behavior is determined after every output query, which may lead to an increase in the number of output queries. The foresight is conducted on the basis of a small subset of already seen values, i.e. the memorable values of the previous node in the tree and the values occurring in the last input, see Algorithm 7.1, line 2. As opposed to this, LearnLib uses all previously seen values. The concrete tree in every cell of the observation table is then used on the one hand to derive memorable values and on the other hand to distinguish between states.

Extensibility As a result of a more modular structure, it is easy to extend the Tomte tool with more functionality. For example, extending the abstraction techniques with other comparisons or operations can be done by merely adapting the mapper. In LearnLib, such a change is more far-reaching, because the entire algorithm has to be changed.

Shared cache In Tomte, the observation tree in the lookahead oracle functions as a cache for repeated output queries sent during hypothesis construction. During hypothesis verification all concrete test sequences are forwarded straight to the SUT without traversing the lookahead oracle and, thus, are not stored. Also in LearnLib a cache component is used, which is placed in between the *learner* and the SUT. However, here it is used in both phases, i.e. during learning and testing. Normalizing all concrete test traces before forwarding them to the cache increases the probability that cached traces can be reused.

7.8.2 Counterexample Analysis

Once a hypothesis is learned, the LearnLib tool as well as the Tomte tool verify if it correctly describes the behavior of the SUT. However, if a counterexample is found, it is handled in a different way by both tools.

Counterexample reduction Whenever a counterexample is found in Tomte, it is shrunk first by eliminating unnecessary cycles as explained in Section 7.4. In LearnLib a counterexample is not optimized before handling. Thus, it is possibly not the shortest one if, e.g., a random test suite is used. This may lead on the one hand to a more expensive processing of the counterexample and on the other hand to a longer suffix that is added to the observation table. The longer suffix in combination with the larger set of values used to determine the future behavior may lead to an increase in the number of output queries.

Counterexample processing As discussed in Section 7.4, processing counterexamples in Tomte is straightforward. A simple test run and check suffice to decide whether a counterexample has to be returned to the *learner* or has to be processed further. In the latter case, the input abstraction has to be refined with a new guard, which can be found by simply analyzing all duplicate values that are not covered by an existing abstraction. Once a guard is found, Tomte does not search for more guards, but updates the input alphabet and restarts learning. Since the lookahead oracle is used as a cache during learning, a new hypothesis can be constructed quickly by only sending sequences of output queries to the SUT that contain the new alphabet symbol. In addition, the previous counterexample is stored and tested again at the beginning of the next testing phase. In contrast, handling counterexamples in LearnLib is more complex. LearnLib iterates stepwise over the counterexample and checks for every step if either a new transition (guard), register (state variable), or location (state) can be found to resolve the counterexample. Moreover, LearnLib continues until it has found all equality guards in the counterexample. Thus, while Tomte only performs computations on some interesting parts of the counterexample to derive a guard, LearnLib analyzes the entire counterexample in detail regarding three conditions. In Tomte, state variables are typically found without any additional effort during the construction of a hypothesis, see Section 7.7.

Minimality Input abstractions in Tomte are applied in every state of the automaton to learn an input-enabled model. However, these abstractions may not be relevant for the behavior of the SUT in every state. Because memorable values can per state be assigned differently to state variables, an abstraction may point to a value that is actually irrelevant. In very rare cases, this may lead to an additional state, meaning that the abstract learned model is not minimal. In contrast, LearnLib always learns the minimal automaton according to the Nerode relation [121].

Using the standardized format and the variety of benchmarks presented in this thesis makes it easy to compare the two algorithms in more detail, e.g. with respect to their limits and the exact number of queries asked for learning and counterexample analysis. An experimental evaluation of both tools on the set of benchmarks can be found in [5]. In addition, this allows us to compare our approach also to other related approaches. This will provide an insight into the strengths and weaknesses of different techniques and enable us to learn from each other. We believe there is still a lot of room for improvement in both tools.

7.9 Conclusions and Future Work

In the last two chapters, we have introduced an approach to automatically infer models of real-world systems that can be represented as scalarset Mealy machines. The algorithm presented in the previous chapter applies counterexample-guided abstraction refinement to derive guards on inputs. In this chapter, we have augmented our algorithm with a lookahead extension, which in combination with a lookahead oracle can be used to deduce the memorable values of an SUT. We have shown the applicability of our approach, amongst others, by inferring models of two communication protocols, the biometric passport, a simple login procedure, and different data structures. In future work, we intend to prove correctness and termination of our approach. The experimental results show that our approach works, but we assume that a few optimizations would improve the overall performance of our Tomte tool. For example, we could improve the handling of lookahead incompleteness. Now, we follow a conservative approach in the sense that we extend the output lookahead trace with one additional symbol. Thus, if an output lookahead trace has to be extended with, for instance, three symbols to find all memorable values, the observation tree detects three times lookahead incompleteness and the entire learning process is also restarted three times. In contrast, we could try to figure out the required output lookahead trace at once. Moreover, we could try to reduce the number of lookahead traces sent from the lookahead oracle to the SUT. A possibility could be to link lookahead traces to certain states of the hypothesis. Similarly, we could also link abstractions to states to reduce the communication with the SUT. In addition, if guards are only added to transitions where they are relevant, this will also simplify the concretization of a hypothesis.

7.A Data Structures

7.A.1 Queue

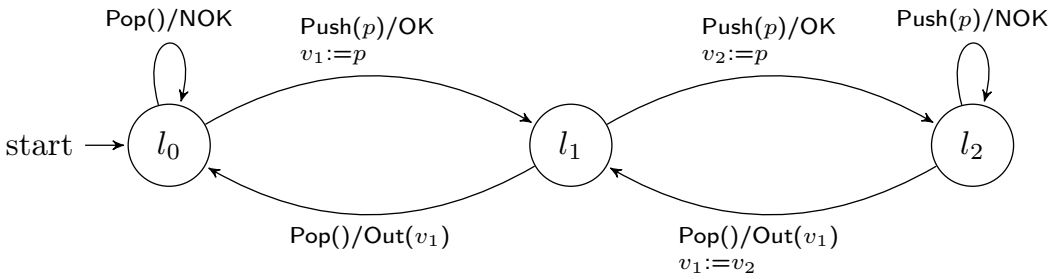


Figure 7.15: *A queue with a capacity of 2 modeled as a scalarset Mealy machine. It is similar to the FIFO-set of Figure 7.12 except that the queue allows to store the same object multiple times.*

7.A.2 2-Dimensional Stack

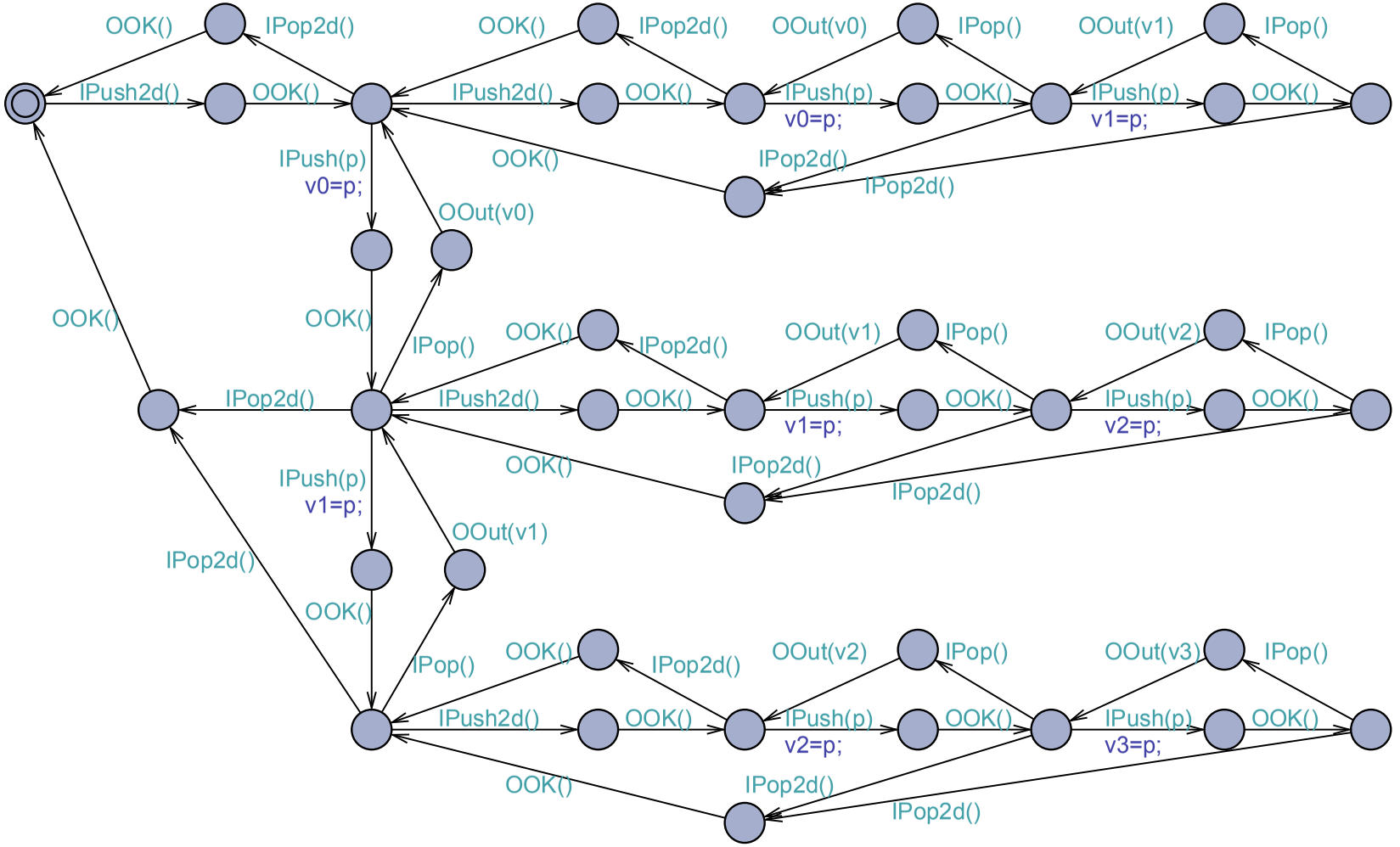


Figure 7.16: *A 2-dimensional stack of overall capacity 4 [80] modeled as a scalarset Mealy machine. Operations* IPush2d *and* IPop2d *operate the outer stack while* IPush *and* IPop *operate the inner stacks. Unsuccessful operations are omitted.*

Part IV

Using Active Learning for Conformance Testing

Chapter 8
Improving Active Mealy Machine Learning for Protocol Conformance Testing

In protocol engineering, protocols are typically specified by independent, authoritative agencies, e.g., the ITU (international telecommunication union) or the IETF (internet engineering task force). For such a protocol specification, a reference implementation is then often developed, which helps to discover errors and ambiguities in the specification, demonstrates that the specification is actually implementable, and serves as a definite interpretation of the specification. Based on the specification and the reference implementation, independent suppliers will develop products implementing the protocol. The reference implementation is then the standard against which these products are measured and compared to assess compliance with the specification.

In this chapter, we investigate a novel application domain for active learning of software models: establishing the correctness of protocol implementations relative to a given reference implementation. To the best of our knowledge, this is a novel application area of regular inference. Moreover, it is a promising one since reference implementations are in existence for many real-world software systems, but models are usually lacking or incomplete.

Our investigation is focused on a well-known benchmark case study from the verification literature: the bounded retransmission protocol [75, 55]. The bounded retransmission protocol is a variation of the classical alternating bit protocol [23] that was developed by Philips Research to support infrared communication between a remote control and a television. We constructed an implementation of the protocol, to which we refer as the reference implementation, and 6 other faulty variations of the reference implementation. Our aim is to combine active learning methods with model-based testing in order to quickly discover the behavioral differences between these variations and the reference. To this aim, we make use of several state-of-the-art tools from regular inference, software testing, and formal verification. We show how these tools can be used for learning models of the bounded retransmission protocol and revealing implementation errors in the mutants.

In addition to experimental results on learning the bounded retransmission protocol, we provide two solutions that significantly reduce the difficulty of answering inclusion queries in this setting:

1. Using abstractions over input and output values through our Tomte learning tool as presented in Chapter 6 and through the model-based testing tool TorXakis [118]. Tomte performs on-the-fly abstractions on concrete input and output values, only introducing new abstract values when required for learning.

2. Comparing a previously learned model of the reference implementation with the current hypothesis using a model equivalence checker (such as the popular CADP model checker [65]). The resulting agent, which we call a *conformance oracle*, effectively transfers knowledge from the reference learning task to the task of learning a mutant, i.e., a slight variation of the reference implementation. The speedup results from the fact that test selection is difficult while equivalence testing is easy.

Our main contributions are demonstrating how active learning can be used in an industrial setting by combining it with software verification and testing tools, and showing how these tools can also be used to analyze and improve the results of learning. The bounded retransmission protocol use case can serve as a benchmark for future active learning and testing methods.

Our research takes place at the interface of model-based testing and model inference, and builds upon a rich research tradition in this area. The idea of combining testing and learning of software systems was first explored by Weyuker, who observed in 1983 "Program testing and program inference can be thought of as being inverse processes" [169]. Recently, there has been much interest in relating model-based testing and model inference in the setting of state machines. Berg et al. [26], for instance, point out that some of the key algorithms that are used in the two areas are closely related. Walkinshaw et al. [167] show that active learning itself is an important source of structural test cases. At the ISoLA 2012 conference a special session was dedicated to the combination of model-based testing and model inference [112], a combination which is often denoted by the term *learning-based testing*. As far as we know, no previous work in this area addresses the problem of conformance with respect to a reference implementation. In addition, the specific combination of tools that we use is new, as well as the case study, and the concept of a conformance oracle.

Organization Section 8.1 gives background information on model-based testing and corresponding MBT tools. Using active learning for establishing conformance of a protocol implementation relative to a given reference implementation is discussed in Section 8.2. Section 8.3 and Appendix 8.A give an overview of the bounded retransmission protocol and the mutants we created. Experimental results on learning models of the bounded retransmission protocol and revealing errors in the mutant implementations are reported in Section 8.4. Section 8.5 presents and analyzes further ideas for improvement. Finally, Section 8.6 concludes our work.

8.1 Model-Based Testing

In *model-based testing* (MBT), which is a new technique that aims to make testing more efficient and more effective [157], the *system under test* (SUT) is tested against a model of its behavior. This model, which is usually developed manually, must specify what the SUT shall do. Test cases can then be algorithmically generated from this model using an MBT tool. When these test cases are executed on the SUT and the actual test outcomes are compared with the model, the result is an indication about compliance of the SUT with the model. Usually, MBT algorithms and tools are *sound*, i.e., a test failure assures non-compliance, but they are not *exhaustive*, i.e., absence of failing tests does not assure compliance: "Program testing can be used to show the presence of bugs, but never to show their absence!" [59].

MBT approaches differ in the kind of models that they support, e.g., state-based models, pre- and post-conditions, (timed) automata, or equational axioms, and in the algorithms that they use for test generation. In this chapter, we concentrate on two state-based approaches: deterministic finite Mealy machines (also called finite state machine (FSM)), and a class of nondeterministic automata, also referred to as labeled transition systems (LTS).

In the Mealy machine approach to MBT, the goal is to test whether a black-box SUT, which is an implementation of an unknown Mealy machine I, is observation equivalent to a given Mealy machine specification S, i.e., to test whether $I \approx S$ [100].

The LTS approach, which does not require determinism, finiteness of states and inputs, input-enabledness, nor alternation of inputs and outputs (a label on a transition is either an input or an output), is more expressive than Mealy machines. Consequently, it requires a more sophisticated notion of compliance between an SUT and a model. The implementation relation **ioco** often serves this purpose [153]. The tools JTorX and TorXakis, among others, generate tests based on this relation; see below. Since it is straightforward to transform a Mealy machine into an LTS, by splitting every (input,output)-pair transition into two LTS transitions with an intermediate state, LTS-based testing can be easily applied to Mealy machine models.

JTorX JTorX [25] is an update of the model-based testing tool TorX [155]. TorX is a model-based testing tool that uses labeled transition systems to derive and execute tests (execution traces) based on **ioco** [153], a theory for defining when an implementation of a given specification is correct. Using on-line testing, TorX can easily generate and execute tests consisting of more than 1 000 000 test events. JTorX is easier to deploy and uses a more advanced version of **ioco**. It contains a graphical user interface for easy configuration, a simulator for guided evaluation of a test trace, interfaces for communication with an SUT, and state-of-the-art testing algorithms.

TorXakis TorXakis [118] is another extension of the TorX model-based testing tool. In addition to the testing algorithms, TorXakis uses symbolic transition

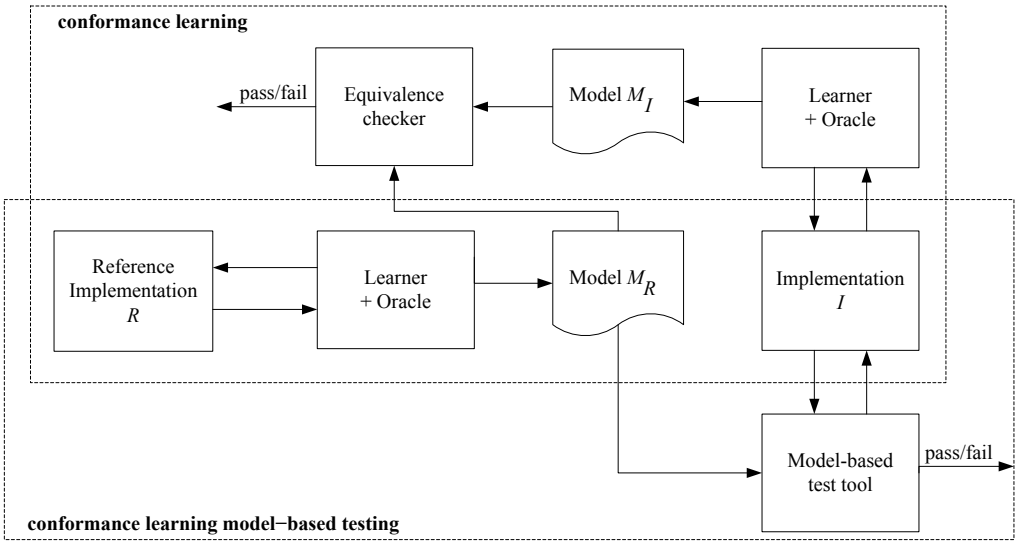


Figure 8.1: *Basic approaches of using automata learning to establish conformance of implementations. The learners interact with the implementations in order to construct models, which are then subsequently used for model-based testing or equivalence checking.*

systems (STSs, LTSs with data variables) with symbolic test generation to deal with structured data, i.e., symbols with data parameters [64], where TorX and JTorX use flattening, i.e., by replacing a parameter by all its possible, necessarily finitely many values. By working symbolically and by exploiting the structure of input actions, TorXakis is able to find certain counterexamples much faster than LearnLib and JTorX.

8.2 Conformance to a Reference Implementation

8.2.1 Conformance Learning and Conformance Model-Based Testing

Figure 8.1 illustrates how we may use automata learning for establishing conformance (i.e., behavior equivalence) of protocol implementations relative to a given reference implementation. Using a state machine synthesis tool, we first actively (query) learn a state machine model M_R of the reference implementation R (using, e.g., LearnLib). Now, given another implementation I, there are basically two things we can do. The first approach, which we call 'conformance model-based testing', is that we provide M_R as input to a model-based testing tool (e.g., JTorX). This tool will then use M_R to generate test sequences and apply them to implementation I in order to establish the conformance of I to the learned model M_R, i.e., whether they implement the same behavior. The model-based testing tool will either output "pass", meaning that the tool has not been able to find any deviating behaviors, or it will output "fail" together with an input sequence that demonstrates the difference between I and M_R. The second, more ambitious approach, which we call 'conformance learning', is to use the learning tool to learn a model M_I of the other implementation I, and then use an equivalence checker to

check observation equivalence of M_R and M_I. The equivalence checker will either output "pass", meaning that the two models are equivalent, or "fail" together with an input sequence that demonstrates the difference between the two models. In the latter case, we check whether this trace also demonstrates a difference between the corresponding implementations R and I. If not, we have obtained a counterexample for one of the two models, which we may feed to the *learner* in order to obtain a more refined model of R or I.

We use the CADP toolset to check observation equivalence of models.

CADP CADP [65] is a comprehensive toolbox for verifying models of concurrent systems, i.e., models consisting of multiple concurrent processes that together describe the overall system behavior. Relying on action-based semantic models, it offers functionalities covering the entire design cycle of concurrent systems: specification, simulation, rapid prototyping, verification, testing, and performance evaluation. It includes a wide range of verification techniques such as reachability analysis and compositional verification. CADP is used in this chapter to check strong bisimulation equivalence of labeled transition systems.

8.2.2 Conformance Learning with a Conformance Oracle

The model-based testing (oracle) part of automata learning can be time consuming in practice. We therefore experimented with an alternative approach in which the model M_R of the reference implementation R is used as an oracle when learning a model for an implementation I. We will see that this use of what we call a *conformance oracle* may significantly speed up the learning process.

Suppose a *learner* has constructed a hypothesized Mealy machine model M_I for implementation I. We want to use the availability of M_R to speed up the validation (or counterexample discovery) for M_I, and reduce the use of the model-based test oracle as much as possible. Our approach works as follows:

1. We first use an equivalence checker to test $M_R \approx M_I$. If so, then we use a model-based test tool to further increase our confidence that M_I is a good model of I. If model-based testing reveals no counterexamples we are done, otherwise the *learner* may use a produced counterexample to construct a new model of I, and we return to step (1).

2. If $M_R \not\approx M_I$ then the equivalence checker produces an input sequence u such that $out_{M_R}(u) \neq out_{M_I}(u)$. We apply u to both implementations R and I, and write $out_R(u)$ and $out_I(u)$, respectively, for the resulting output sequences.

3. If $out_R(u) \neq out_{M_R}(u)$ then model M_R is incorrect and we may use counterexample u to construct a new model for R.

4. Otherwise, if $out_I(u) \neq out_{M_I}(u)$ then model M_I is incorrect. In this case the *learner* may use counterexample u to construct a new model for I, and we return to step (1).

5. Otherwise, we have identified an observable difference between implementations R and I, i.e., I is not conforming to reference implementation R.

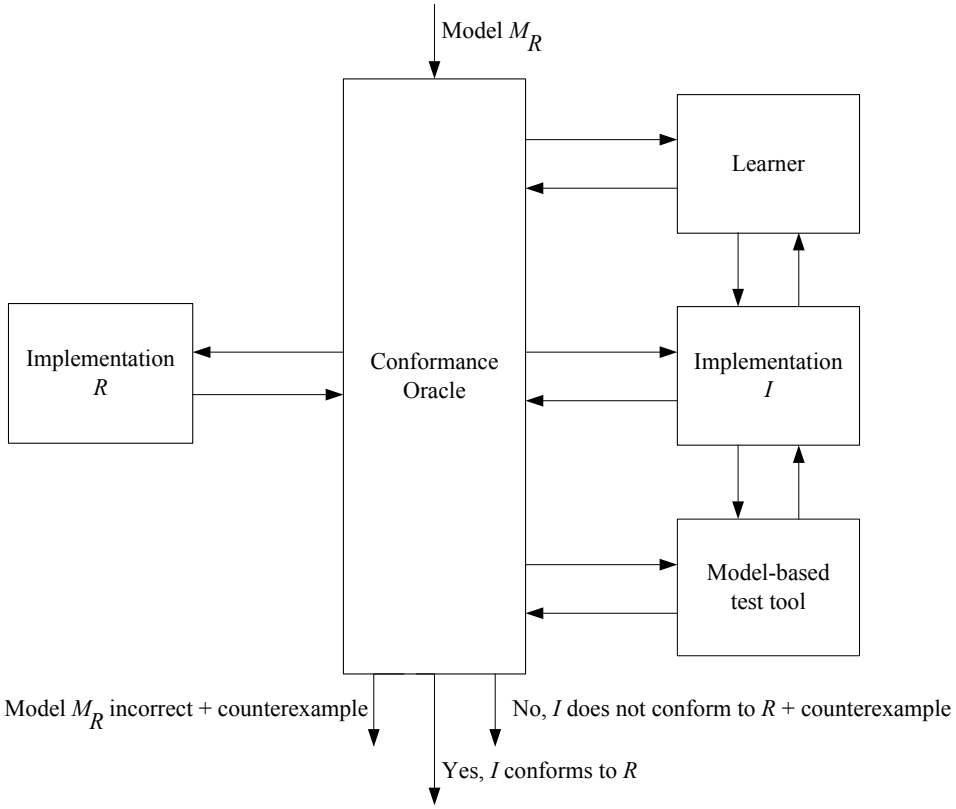


Figure 8.2: *Conformance oracle. The oracle takes a previously learned model M_R for the reference R as input and uses it to quickly answer equivalence queries by testing the equivalence of M_R with the hypothesized model M_I for the implementation I. When they are found to be inequivalent, the discovered counterexample is provided as input to I and R in order to test whether it proves the inequivalence of R and I, M_I and I, or M_R and R.*

Figure 8.2 illustrates the architectural embedding of a conformance oracle.

8.3 The BRP Implementation and its Mutants

The bounded retransmission protocol (BRP) [75, 55] is a variation of the well-known alternating bit protocol [23] that was developed by Philips Research to support infrared communication between a remote control and a television. In this section, we briefly recall the operation of the protocol, and describe the reference implementation of the sender and the 6 mutant implementations.

The bounded retransmission protocol is a data link protocol which uses a stop-and-wait approach known as 'positive acknowledgement with retransmission': after transmission of a frame the sender waits for an acknowledgement before sending a new frame. For each received frame the protocol generates an acknowledgement. If, after sending a frame, an acknowledgement fails to appear, the sender times out and retransmits the frame. An alternating bit is used to detect duplicate transmission of a frame.

Figure 8.3 illustrates the operation of our reference implementation of the sender of the BRP. We use the graphical user interface of the model-checker Up-

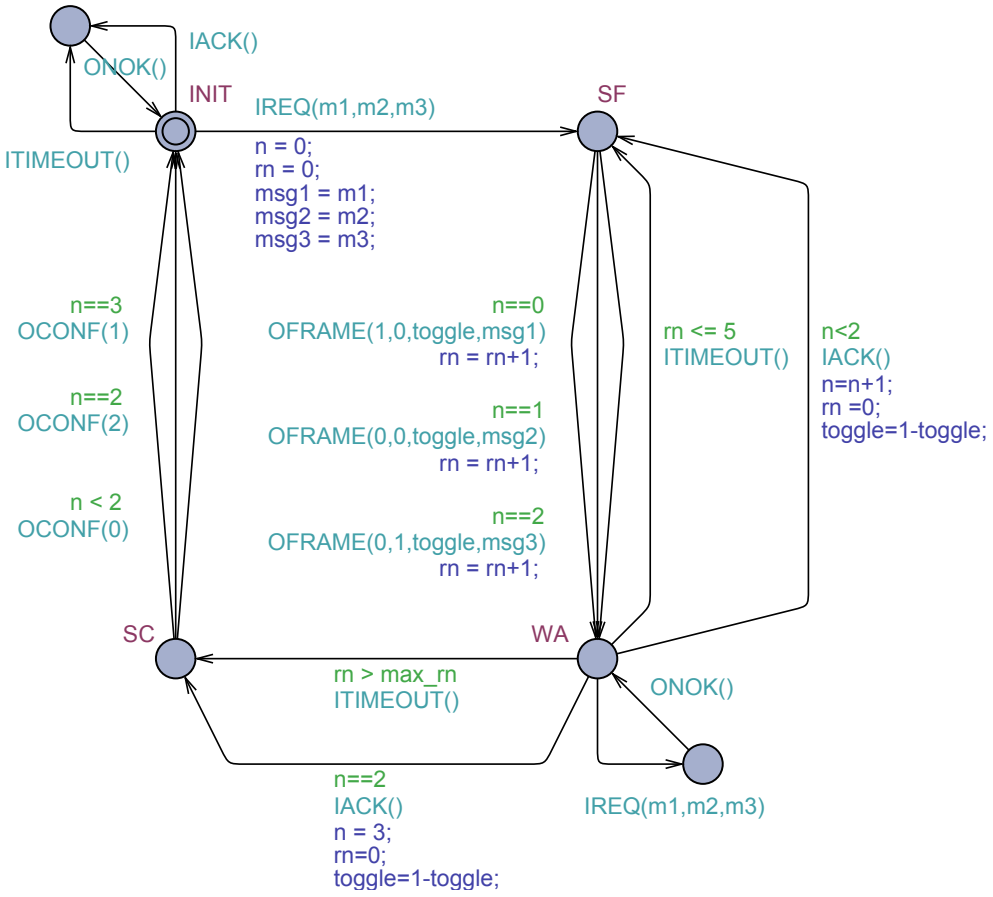


Figure 8.3: *Reference implementation of the BRP sender. The input symbols start with I, the output symbols start with O. In addition to symbols, the transitions contain value checks (or guards, ==, <, >) and assignments (=).*

paal [24] as an editor for extended finite state machines. The reference implementation that we used is a Java executable that was generated automatically from this diagram. The sender protocol uses the following inputs and outputs:

- Via an input IREQ(m_1, m_2, m_3), the upper layer requests the sender to transmit a sequence m_1 m_2 m_3 of messages. For simplicity, our reference implementation only allows sequences of three messages, and the only messages allowed are 0 and 1. When the sender is in its initial state INIT, an input IREQ(m_1, m_2, m_3) triggers an output OFRAME(b_1, b_2, b_3, m), otherwise it triggers output ONOK.

- Via an output OFRAME(b_1, b_2, b_3, m), the sender may transmit a message to the receiver. Here m is the actual transmitted message, b_1 is a bit that is 1 iff m is the first message in the sequence, b_2 is a bit that is 1 iff m is the last message in the sequence, and b_3 is the alternating bit used to distinguish new frames from retransmissions.

- Via input IACK the receiver acknowledges receipt of a frame and via input ITIMEOUT the sender is informed that a timeout has occurred, due to the loss of either a frame or an acknowledgement message. When the sender is in state WA ("wait for acknowledgement"), an input IACK or ITIMEOUT

triggers either an output OFRAME(b_1, b_2, b_3, m) or an output OCONF(i). If the sender is not in state WA, ONOK is triggered.

- Via an output OCONF(i), the sender informs the upper layer about the way in which a request was handled:
  - $i = 0$: the request has not been dispatched completely,
  - $i = 1$: the request has been dispatched successfully,
  - $i = 2$: the request may or may not have been handled completely; this situation occurs when the last frame is sent but not acknowledged.

  An output OCONF occurs when either all three messages have been transmitted successfully, or when a timeout occurs after the maximal number of retransmissions.

Note that, within the state machine of Figure 8.3, inputs and outputs strictly alternate. Thus it behaves like a Mealy machine. The state machine maintains variables msg1, msg2 and msg3 to record the three messages in the sequence, a Boolean variable toggle to record the alternating bit, an integer variable n to record the number of messages that have been acknowledged, and an integer variable rn to record the number of times a message has been retransmitted. Each message is retransmitted at most 5 times.

We consider the following six mutants of the reference implementation of the sender (see Appendix 8.A):

1. Whereas the reference implementation only accepts a new request in the INIT state, mutant 1 also accepts new requests in state WA. Whenever mutant 1 receives a new request, the previous request is discarded and the sender starts handling the new one.
2. Whereas in the reference implementation each message is retransmitted at most 5 times, mutant 2 retransmits at most 4 times.
3. Whereas in the reference implementation the alternating bit is only toggled upon receipt of an acknowledgement, mutant 3 also toggles the alternating bit when a timeout occurs.
4. In mutant 4 the first and last control bit for the last message are swapped.
5. Mutant 5 outputs an OCONF(0) in situations where the reference implementation outputs OCONF(2).
6. If the first and the second message are equal then mutant 6 does not transmit the third message, but instead retransmits the first message.

Since input and output messages still alternate, all of the mutants still behave as Mealy machines. For all BRP implementations, we consider the inputs: IREQ(m_1, m_2, m_3), IACK, and ITIMEOUT, where m_1, m_2, and m_3 can be either 0 or 1. Thus, the input alphabet consists of 10 input symbols: 8 different IREQ inputs, one IACK input, and one ITIMEOUT input. We also have the following outputs: ONOK, OFRAME(b_1, b_2, b_3, m), and OCONF(i), where $0 \leq i \leq 2$, i.e., 20 output symbols. In the next section, we discuss how to connect these implementations to an active Mealy machine *learner* and a model-based testing tool.

8.4 Experiments

In this section, we report on the experiments that we did using LearnLib and JTorX to establish conformance of the six mutant implementations to the reference implementation.

Learning BRP models In order to learn models of the reference implementation and its mutants, we connect the implementations, which serve as SUT, to the LearnLib tool.[1] Since all BRP implementations behave as Mealy machines, this is the type of state machine that we infer with our approach. In our experiments we consider the inputs: IREQ(m_1, m_2, m_3), IACK, and ITIMEOUT, where m_1, m_2 and m_3 can be either 0 or 1. Thus, the input alphabet consists of 10 input symbols: 8 different IREQ inputs, one IACK input, and one ITIMEOUT input. Moreover, we have the following outputs: ONOK, OFRAME(b_1, b_2, b_3, m), where b_1, b_2, b_3 and m can be either 0 or 1, and OCONF(i), where $0 \leq i \leq 2$. In order to approximate equivalence queries, we used the LearnLib test suite with randomly generated test traces containing 100 inputs.

The results of the inference of the reference implementation and the six mutants are shown in Table 8.1. For every implementation, we list the number of states in the learned model, as well as the average total number of output queries (OQ). Moreover, we list the average total number of test traces (TT) generated for approximating equivalence queries. Note that these numbers do not include the last equivalence query, in which no counterexample has been found. Using CADP, we verified that all the learned models indeed are correct, i.e., equivalent to the Uppaal models described in Section 8.3. Each experiment was repeated 10 times with different seeds for the equivalence queries. For each measured value its average over the 10 experiments is listed in the table together with the standard deviation. If an experiment did not succeed to learn the model within two hours we aborted the experiment. In the last row we display how many of the 10 experiments did succeed. Even if an experiment did fail, we still use its numbers in calculating the average and the standard deviation, giving a lower bound for the real average and standard deviation for this experiment.

If we take a closer look at Table 8.1, we observe some interesting peculiarities. First, the number of test traces for mutant 1 is much higher than for the other implementations. The reason for this is that mutant 1 also accepts new

[1] In previous work [9] we used TCP/IP socket communication. TCP/IP uses optimizations, TCP delayed acknowledgment technique and Nagle's algorithm, for reducing packet overhead. These optimizations slow down the communication in a setting with very small messages. In normal communication between SUT and *learner* the communication is an alternating pattern of sending input followed by a returned output. However in between two queries to the SUT an extra RESET input is sent, which breaks this alternating pattern. Exactly at that point the optimizations in TCP/IP cause a delay in communication to happen. Effectively this means that *each* query to the SUT gets an extra delay. By disabling the Nagle optimization in the TCP/IP socket communication we can prevent these delays to happen. For even better performance we used direct method calls to the SUT by linking the SUT code against our *learner*. Removing this delay made the queries in the experiments much faster and therefore it was possible to learn the BRP models for larger retransmission counter and variable ranges in the same amount of time than in [9]. Moreover because output queries in general are much shorter than test traces, the performance gain per query in the first are much bigger than in the latter.

		RefImpl	Mut1	Mut2	Mut3	Mut4	Mut5	Mut6
vr0-1	states	156	156	128	156	156	156	136
	avg. OQ	24998	22174	19618	22488	24684	24998	21662
	std.dev.	1717	627	1006	769	1828	1717	736
	avg. TT	14	5830	7	15	14	14	14
	std.dev.	13	5142	6	13	13	13	13
	Succeeded	10/10	10/10	10/10	10/10	10/10	10/10	10/10

Table 8.1: *Learning statistics for the BRP reference implementation and mutants 1-6.* ***OQ*** *refers to the average total number of output queries,* ***TT*** *to the average total number of test traces (in both cases sequences of inputs have been counted).* ***vrX-Y*** *means $m_i \in [X...Y]$, where m_i is a message in* IREQ(m_1, m_2, m_3)*, e.g. vr0-1 allows values 0 and 1 in a message in* IREQ*.*

requests in state WA. Whenever mutant 1 receives a new request, the previous request is discarded and the sender starts handling the new request. This makes it much harder to find a counterexample that produces an OCONF(0) or OCONF(2) output, since this requires six successive ITIMEOUT inputs without intermediate IREQ inputs. The probability that LearnLib selects (uniformly at random) six successive ITIMEOUT inputs in a row is low, since each time ITIMEOUT only has a 10% chance of being selected. This issue will be analyzed in more detail in Section 8.5. Second, the numbers for mutant 2 are slightly smaller than for the other implementations. The reason for this is that in mutant 2 the maximal number of retransmissions is smaller: 4 instead of 5, see Figures 8.4 - 8.9. The size of the model and the times required for constructing and testing hypotheses (explored in the next section) all depend on the maximal number of retransmissions. This will be explored further in the next section.

More learning experiments Besides the maximal value of the retransmission counter, also changes in the domain of message parameters m_1, m_2, and m_3 will influence the learning results for the different implementations, because more inputs need to be considered. Therefore, we ran some additional experiments for different parameter settings of the reference implementation and mutant 1 (the behavior of mutants 2-6 is similar to that of the reference implementation, because no changes to the general structure of handling requests have been made). We evaluated how LearnLib performs for different maximal values for the retransmission counter **rn**. Moreover, we investigated what happens when increase the value range **vr** for each message parameter.

Table 8.2 and 8.3 show the results of learning models of the reference implementation and mutant 1 using different maximal numbers of retransmission and different value ranges for the message parameters m_1, m_2, and m_3. As expected, increasing the number of retransmissions **rn** and the value ranges **vr** both results in bigger models for which more output and test traces are needed and, accordingly, more time for learning and testing. Increasing the value range leads to a fast growth in output queries required to construct a hypothesis and the number of test traces required to find counterexamples for incorrect hypotheses, because the model to learn contains more states and the probability to select an ITIMEOUT

——— increasing retransmission counter —>

		rn15	rn16	rn17	rn18	rn19	rn20
vr0-1	states	436	464	492	520	548	576
	avg. OQ	83346	91949	99465	107206	108582	104113
	std.dev.	4122	5807	4298	13624	6402	33896
	avg. TT	20699	82854	103041	137750	295292	2665735
	std.dev.	21865	129195	130379	180024	284401	3044574
	Success	10/10	10/10	10/10	10/10	10/10	9/10
		rn09	**rn10**	**rn12**	**rn13**	**rn14**	**rn15**
vr0-2	states	730	808	964	1042	1120	1198
	avg. OQ	755837	853000	1037250	1124192	1211588	564335
	std.dev.	13562	18753	31762	37755	29241	603545
	avg. TT	7507	20944	348407	1349612	2395184	7903428
	std.dev.	5219	14340	460829	1882704	2608484	2833324
	Succeeded	10/10	10/10	10/10	10/10	10/10	4/10
		rn06	**rn07**	**rn08**	**rn09**	**rn10**	**rn11**
vr0-3	states	1052	1220	1388	1556	1724	1892
	avg. OQ	4971425	5789485	6641683	7445564	5975070	2433575
	std.dev.	34020	84075	84468	105242	3413383	3341094
	avg. TT	2587	14928	111122	502686	3795812	8129711
	std.dev.	1682	9797	116094	545382	3528212	3376705
	Succeeded	10/10	10/10	10/10	10/10	7/10	2/10
		rn03	**rn04**	**rn05**	**rn06**	**rn07**	**rn08**
vr0-4	states	994	1304	1614	1924	2234	2544
	avg. OQ	16537318	21810721	27159739	32278521	37677908	27794859
	std.dev.	56460	106048	102494	171051	277996	18521581
	avg. TT	230	1392	13057	148571	1375285	4732741
	std.dev.	246	1089	13744	162899	1056963	3445873
	Succeeded	10/10	10/10	10/10	10/10	10/10	6/10
		rn02	**rn03**	**rn04**	**rn05**	**rn06**	
vr0-5	states	1120	1636	2152	2668	3184	
	avg. OQ	53959581	79140421	104476825	129818719	130303196	
	std.dev.	0	106997	214990	232653	43041807	
	avg. TT	106	1353	16010	98888	2591811	
	std.dev.	77	1287	11680	116544	1743665	
	Succeeded	10/10	10/10	10/10	10/10	6/10	
		rn02	**rn03**	**rn04**			
vr0-6	states	1712	2510	3308			
	avg. OQ	205602133	302303504	353391698			
	std.dev.	177195	259788	51494449			
	avg. TT	294	6044	105437			
	std.dev.	378	5491	65300			
	Succeeded	10/10	10/10	5/10			

——— increasing value range —>

Table 8.2: *Learning statistics for reference implementation.* ***OQ*** *refers to the average total number of output queries,* ***TT*** *to the average total number of test traces (in both cases sequences of inputs have been counted).* ***vrX-Y*** *means $m_i \in [X...Y]$, where m_i is a message in* IREQ(m_1, m_2, m_3)*, e.g. vr0-3 allows values 0,1,2, and 3 in a message in* IREQ*. Similar,* ***rn*** *refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement (rn>5 and rn≤5) for the* ITIMEOUT *input from state* WA *to* SF *and from* WA *to* SC*.*

		——— increasing retransmission counter ——>					
		rn03	**rn04**	**rn05**	**rn06**	**rn07**	**rn08**
vr0-1	states	100	128	156	184	212	240
	avg. OQ	13214	18199	22174	27999	32256	27365
	std.dev.	402	515	627	738	850	16821
	avg. TT	154	1250	5830	61495	667761	7903190
	std.dev.	129	1615	5142	78465	516786	3371144
	Succeeded	10/10	10/10	10/10	10/10	10/10	6/10
		rn02	**rn03**	**rn04**	**rn05**		
vr0-2	states	184	262	340	418		
	avg. OQ	171318	243928	326408	334443		
	std.dev.	1603	2281	2960	131246		
	avg. TT	347	9655	278077	4388168		
	std.dev.	331	8693	159918	2974071		
	Succeeded	10/10	10/10	10/10	8/10		
		rn02	**rn03**	**rn04**			
vr0-3	states	380	548	716			
	avg. OQ	1730589	2495661	1272022			
	std.dev.	0	0	1017988			
	avg. TT	3312	224096	6522672			
	std.dev.	3031	246072	2488837			
	Succeeded	10/10	10/10	2/10			

——— increasing value range ———>

Table 8.3: *Learning statistics for mutant 1.* ***OQ*** *refers to the average total number of output queries,* ***TT*** *to the average total number of test traces (in both cases sequences of inputs have been counted).* ***vrX-Y*** *means $m_i \in [X...Y]$, where m_i is a message in* IREQ(m_1, m_2, m_3)*, e.g. vr0-3 allows values 0,1,2, and 3 in a message in* IREQ*. Similar,* ***rn*** *refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement (rn>5 and rn≤5) for the* ITIMEOUT *input from state* WA *to* SF *and from* WA *to* SC*.*

or IACK input (which is needed in a counterexample) shrinks. For example, compare the results for rn15 and vr0-1 versus rn15 and vr0-2 in Table 8.2. Increasing the maximal number of retransmissions leads to a linear growth of output queries required to construct a hypothesis, but to a fast growth of test traces required to find counterexamples for incorrect hypotheses, because the higher the retransmission counter, the more ITIMEOUT inputs are needed in a counterexample. For example, compare the results for rn15 to rn20 for vr0-1 in Table 8.2.

For mutant 1, the time needed for testing increases so fast that if the maximal number of retransmissions is 8 and there are 2 values, not every seed results in a correct model within 2 hours. In contrast, for 2 values the reference implementation only starts failing to learn the correct model within hours for all seeds for the maximal number of retransmissions of 20. As mentioned before, the reason for this is that mutant 1 also accepts new requests in state WA and discards previous requests. Also in the case where the maximal number of retransmissions is 5 and there are 3 values, LearnLib is not able to construct a correct model for all seeds for mutant 1 within 2 hours. This is not surprising, because in both cases the probability to select a counterexample is even lower than for mutant 1 in Table 8.1. Once LearnLib fails to learn a correct model, we assume that it will

also fail for larger value ranges of the parameters. The same holds for the retransmission counter. Once some of the experiments fail, the number of successful runs with a higher retransmission counter will decrease.

Conformance checking We compare the two methods, described in Section 8.2.1, for establishing the conformance of the mutant implementations to the reference implementation of BRP. We only consider the versions of the models with at most 5 retransmissions and 2 different parameter values.

The first method, conformance learning, used the CADP (bisimulation) equivalence checker to compare the model M_I, that we learned for the mutant implementations I, with the model M_R learned for the reference implementation R.[2] For each of the mutants, CADP quickly found a counterexample trace illustrating the difference between the models of the mutant and the model of the reference implementation (for each mutant it takes around 3 seconds to find the counterexample). The counterexamples found by CADP are depicted in Table 8.4.

	counterexample	**output**	**expected**
Mut1	IR(0,0,0) IR(0,0,0)	OF(1,0,0,0)	ONOK()
Mut2	IR(0,0,0) IT() IT() IT() IT() IT()	OCONF(0)	OF(1,0,0,0)
Mut3	IR(0,0,0) IT()	OF(1,0,1,0)	OF(1,0,0,0)
Mut4	IR(0,0,0) IA() IA()	OF(1,0,0,0)	OF(0,1,0,0)
Mut5	IR(0,0,0) IA() IA() IT() IT() IT() IT() IT() IT()	OCONF(0)	OCONF(2)
Mut6	IR(0,0,1) IA() IA()	OF(0,1,0,0)	OF(0,1,0,1)

Table 8.4: *Counterexamples found by equivalence checking of mutant models and reference implementation (IT = ITIMEOUT, IR = IREQ, IA = IACK, OF = OFRAME).*

The second method, conformance model-based testing, used the model M_R of the reference implementation R as input for the JTorX model-based testing tool and the mutant implementations I as SUTs. Test steps were executed until a counterexample was found. Again, JTorX found a counterexample for each of the mutant implementations. The average number of IO symbols, i.e. the average number of inputs and outputs in the counterexample, is shown in Table 8.5 for the different mutant implementations. Because JTorX generates a single long testing sequence without intermediate resets, the resulting counterexamples are rather long sequences and therefore are not shown in the table.

In a JTorX experiment the performance is measured in number of IO symbols whereas in a LearnLib experiment the performance is measured in number of output and testing queries. Comparing the measurements of both experiments is not trivial. Although each testing query corresponds to 200 IO symbols, since we use a random test suite generating test traces with 100 input symbols, each output query varies in length determined by the learning algorithm. However, in a crude approximation we could say that in the learning experiment $\#IOsymbols \approx x * OQ + 200 * TT$ where x is a guessed factor of the average length of the output queries.

[2]Essentially the same counterexamples were also found using the JTorX iocoChecker. The JTorX iocoChecker, which is distributed as an extra feature of the JTorX model-based testing

		mut1	mut2	mut3	mut4	mut5	mut6
vr0-1	avg. IO symbols	5	894	19	37	6657	198
rn05	std.dev.	1	768	12	20	4983	268

Table 8.5: *Conformance testing the mutants with JTorX for retransmission counter 5 and value range 0-1: using the reference model JTorX will generate a test sequence and applies it to the mutant implementation in order to establish the conformance between the reference model and the mutant. For all mutants it detected the non-conformance and returned a counterexample. The table shows the average number of IO symbols, i.e. the average number of inputs and outputs, needed to find the counterexample for each mutant.*

If we look at Table 8.5 we immediately see that the number of IO symbols needed for mutant 5 is much bigger than for the other mutants. However, Table 8.1 shows that learning mutant 5 is not more difficult than learning the other mutants. This can be explained by the fact that if we look at the counterexample for mutant 5 in Table 8.4 we immediately see that the counterexample for mutant 5 is the longest. When looking at the model for mutant 5 we can see that the number of ITIMEOUT inputs in the counterexample is directly related to the value of the retransmission counter **rn**.

To investigate the effects, we increase the retransmission counter **rn** to 10. The learning results for $rn = 10$ are shown in Table 8.6 for the different mutants and for conformance testing the mutants with JTorX in Table 8.7. Note that in learning the mutants for the retransmission counter of 10 we skipped mutant 1, because it was already shown in Table 8.1 that it couldn't be learned in 2 hours, and we skipped mutant 2 because the only difference between mutant 2 and the reference implementation is a different retransmission number.

		RefImpl	Mut3	Mut4	Mut5	Mut6
vr0-1	states	296	296	296	296	256
rn10	avg. OQ	55417	46509	55417	55417	47166
	std.dev.	3767	1899	3767	3767	3083
	avg. TT	280	357	280	280	280
	std.dev.	171	266	171	171	171
	Succeeded	10/10	10/10	10/10	10/10	10/10

Table 8.6: *Conformance checking using the equivalence checker CADP for retransmission counter 10 and value range 0-1.* ***OQ*** *refers to the average total number of output queries,* ***TT*** *to the average total number of test traces (in both cases sequences of inputs have been counted). The table shows the learning statistics for reference implementation and mutants 3-6. After learning the reference and mutant models we could find in around 3 seconds a counterexample trace illustrating the difference between the model of the mutant and the model of the reference implementation.*

After learning the new mutants CADP could again find in around 3 seconds a counterexample trace illustrating the difference between the models of the mutants

tool [25], is a tool to check analytically whether two models are ioco-related.

		mut1	mut3	mut4	mut5	mut6
vr0-1	avg. IO symbols	5	19	37	109154	178
rn10	std.dev.	1	12	20	90617	228

Table 8.7: *Conformance testing the mutants with JTorX for retransmission counter 10 and value range 0-1: using the reference model JTorX will generate a test sequence and applies it to the mutant implementation in order to establish the conformance between the reference model and the mutant. For all mutants it detected the non-conformance and returned a counterexample. The table shows the average number of IO symbols, i.e. the average number of inputs and outputs, needed to find the counterexample for each mutant.*

and the model of the reference implementation. The results in Table 8.6 show that learning mutant 5 with a larger retransmission counter still is not more difficult than learning the other mutants, whereas Table 8.7 shows that for mutant 5 model-based testing still requires much longer traces than for the other mutants. However, when comparing Table 8.6 and Table 8.7 it shows that for all mutants the number of IO symbols is only increased for mutant 5.

Again, this can be explained by the fact that if we look at the counterexample for mutant 5 in Table 8.4 we immediately see that the number of ITIMEOUT inputs in the counterexample is directly related to the value of the retransmission counter **rn**. By increasing the retransmission counter **rn**, the counterexample for mutant 5 becomes longer, and since finding longer counterexamples takes longer, this increases the time required by JTorX to find a counterexample. In contrast, for mutants 1,3,4 and 6 the counterexamples have no ITIMEOUT inputs and are independent of the retransmission counter **rn**, which explains why in the experiments for these mutants the number of IO symbols for finding counterexamples doesn't increase for a larger retransmission counter **rn**.

Thus by comparing Table 8.6 and Table 8.7 we see that the increase in the number of IO symbols needed for conformance testing of mutant 5 is big. However, this number of IO symbols needed for conformance testing of mutant 5 is comparable in size to the number of output and testing queries for learning mutant 5: compare 109154 IO symbols from JTorX to $111417 = 1 \cdot 55417 + 200 \cdot 280$ ($\#IOsymbols \approx x \cdot OQ + 200 \cdot TT$) IO symbols from LearnLib, where we assume a lower bound $x = 1$ on the average length of output queries.

Since LearnLib tests hypotheses in a way similar to JTorX and because the counterexample of mutant 5 reaches the deepest part of the model, testing during learning the model with LearnLib of mutant 5 and conformance testing with JTorX of mutant 5 become even difficult. Therefore their performance in both experiments becomes comparable.

In general, however, learning models of proposed implementations takes more time than model-based testing them but also provides more information in the form of a learned model. Even in these cases, it may still be beneficial to use learning tools since the learned models can for instance be used for model checking analysis.

8.5 Further Analysis and Improvements

8.5.1 Why Random Testing Sometimes Fails

In our experiments, the most effective technique available in LearnLib for approximating equivalence queries turned out to be random testing. In order to analyze the effectiveness of this method, we may compute the probabilities of reaching states that provide a counterexample within a certain number of transitions, by translating the Mealy machine of the *teacher* (the system under test) into a discrete time Markov chain (DTMC). This DTMC has the same states as the Mealy machine, and the probability of going from state q to state q' is equal to the number of transitions from q to q' in the Mealy machine divided by the total number of inputs. Through analysis of this DTMC, the MRMC model checker can compute the probability of finding certain counterexamples within a given time.

MRMC MRMC [95] is a probabilistic model checker, which can be used to check the probability that a logical statement (such as a system breakdown) occurs in a given continuous- or discrete-time Markov chain, with or without reward functions (common in Markov decision processes [127]). Such a logical statement can be expressed in a probabilistic branching-time logic PCTL [74] or CSL [17]. The probabilistic models may also contain reward functions and bounds on these can be checked in combination with the time and probability values.
We use MRMC to compute the probability of reaching certain states in an implementation within a certain number of steps in a setting where inputs are generated randomly. We wrote a small script that converts LTSs in `.aut` format to DTMCs in `.tra/.lab` format, which are accepted as input by MRMC.
Using MRMC we computed that for the reference implementation with up to 7 retransmissions the probability of reaching, within a single test run of 125 steps, a state with an outgoing OCONF(0) transition is 0.0247121. This means the probability of reaching a state with an outgoing OCONF(0) transition within 75 test runs is 0.847. This result explains why LearnLib requires very few test runs to learn a correct model of this system, which it does within seconds. Using MRMC, we also computed that for the version of mutant 1 with up to 7 retransmissions the probability of reaching, within 125 steps, a state with an outgoing OCONF(0) transition is only 0.0000010. Hence, finding a counterexample by random testing will take much longer for mutant 1 than for the reference, explaining why LearnLib needs 667761 (see Table 8.3) test runs on average to find this counterexample ($0.999999^{667761} \approx 0.51$ is the probability of not finding this trace in 667761 tries).

8.5.2 Using Abstraction to Speed Up Testing

A simple way to increase the probability of finding counterexamples for mutant 1 within a short time is to increase the probability of ITIMEOUT inputs, for instance by assigning an equal probability of 1/3 to the ITIMEOUT, IACK and IREQ inputs. Symbolic testing tools and abstraction learners will use this distribution, because they initially assume that the parameter values are not as important as the input types. TorXakis and Tomte (see Section 8.1 and 6.1) are such tools and

we therefore evaluated the effect of using these instead of the basic random testing implemented in LearnLib.

TorXakis For the experiments above we employed LearnLib for both the learning and testing phase. However, it is also possible to perform the equivalence check by an external tool. We experimented with the LTS-based model-based testing tool TorXakis to see whether we can improve on detecting incorrect hypothesis models.

		rn5	**rn7**
vr0-1	(avg. #test symbols, st. dev)	(1785, 1921)	(19101, 22558)
vr0-2	(avg. #test symbols, st. dev)	(2991, 2667)	(18895, 15158)
vr0-9	(avg. #test symbols, st. dev)	(3028, 3199)	

Table 8.8: *Equivalence query statistics for mutant 1 with TorXakis.* ***vrX-Y*** *means* $m_i \in [X...Y]$*, where* m_i *is a message in* IREQ(m_1, m_2, m_3)*, e.g. vr0-2 allows values 0,1, and 2 in a message in* IREQ*. Similar,* ***rn*** *refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement (rn>5 and rn≤5) for the* ITIMEOUT *input from state* WA *to* SF *and from* WA *to* SC*.*

Table 8.8 summarizes the results obtained with TorXakis when testing mutant 1 against the hypothesized LearnLib model for $rn5, vr0-1$, $rn7, vr0-1$ and $rn5, vr0-2$. In addition, the results for the scenarios $rn5, vr0-9$ and $rn7, vr0-2$ are presented. LearnLib did not manage to find a counterexample in these cases. The numbers in Table 8.8 are the average lengths of the test runs, measured in test symbols (both input and output), until a discrepancy between the model and mutant 1 was detected. The average is taken over 10 different random test runs. We do not measure the timing, because it makes no sense for TorXakis: TorXakis explicitly tests for non-occurrence of outputs by means of a time-out, and while the time value chosen for this time-out is in some sense arbitrary, it has a very strong influence on the total duration of a test.

TorXakis is able to detect counterexamples for the incorrect hypothesized models for $rn5, vr0-1$, $rn7, vr0-1$, and for $rn5, vr0-2$. Only after the IREQ input has been selected, the message values are randomly selected. This implies that increasing the domain of possible message values does not increase the length of the test case required to detect the counterexample. Combined with the fact that TorXakis generates one very long test case, it is able to find a counterexample for the scenario $rn7, vr0-2$ within reasonable time. In conclusion, TorXakis is able to detect counterexamples within reasonable time, which LearnLib could not detect.

Tomte Through the use of counterexample abstraction refinement, Tomte is able to learn models for a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. The version of Tomte used for the experiments in this chapter is based on Chapter 6, where only the first and the last occurrence of parameters of actions are remembered. As a result, we could only learn models such as mutant 1, where

		rn09	rn10	rn12	rn13	rn14	rn15
vr0-∞	states	62	68	80	86	92	98
	avg. OQ	1518	1875	2449	2629	2196	1377
	std.dev.	3	4	3	4	1372	1564
	avg. TT	1175	3525	51589	89133	182508	258552
	std.dev.	1998	2816	34687	33648	82534	78432
	Succeeded	10/10	10/10	10/10	10/10	7/10	4/10

Table 8.9: *Learning statistics for mutant 1 using Tomte.* ***vrX-Y*** *means $m_i \in [X...Y]$, where m_i is a message in* IREQ*(m_1, m_2, m_3), e.g. vr0-1 allows values 0 and 1 in a message in* IREQ*.* ***rn*** *refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($rn>5$ and $rn\leq 5$) for the* ITIMEOUT *input from state* WA *to* SF *and from* WA *to* SC*.*

each IREQ input overwrites previous occurrences of the message parameters. For these instances, however, Tomte outperforms LearnLib with several orders of magnitude. Table 8.9 gives an overview of the statistics for learning mutant 1 with Tomte. Since in Tomte the entire range of message values for mutant 1 is abstracted into a single equivalence class, Tomte needs far fewer queries than LearnLib (cf. Table 8.3) and, moreover, Tomte can handle a larger value range than LearnLib: compare the results vr0-∞ used by Tomte to vr0-3 used by LearnLib. With the extension of Chapter 7, we can also learn the BRP reference implementation and the other mutants as presented in Section 8.3, see Table 7.3 for the learning statistics. However, we did not repeat the experiments with increased values of the retransmission counter.

8.5.3 Conformance Learning with a Conformance Oracle

Above, we demonstrated how to make random testing a little smarter by making use of abstractions. This results in a speed up in the time random testing requires to find a counterexample, but even this method has its limits as shown in Table 8.9. We now show that we can remove random testing altogether using the notion of a conformance oracle using the method 'Conformance learning with a conformance oracle' described in Section 8.2.2, in which we use CADP as an equivalence checker.

		mut1	mut2	mut3	mut4	mut5	mut6
vr0-1	states	16	127	16	16	155	20
rn05	OQ	1771	17798	1771	1771	21718	2211

Table 8.10: *Finding counterexamples using a conformance oracle, case* ***rn***$= 5$*. In fact we are learning the mutant using the reference model as a conformance oracle until we found a counterexample which proves both the learned mutant model and the mutant implementation different from the reference implementation.*

The results of our experiments can be found in Table 8.10 in which for each mutant we found non-conformance with the reference implementation. The number of states listed are the number of states of the last hypothesis found in step

5 of the conformance learning with a conformance oracle algorithm described in Section 8.2.2. Compared with Table 8.9, we only have the number of states and the number of output queries listed, because when using a conformance oracle we do no random testing after learning anymore. Thus, by using a conformance oracle instead of random testing, the learning process becomes completely deterministic and, therefore, we only had to do each experiment in Table 8.10 once.

Table 8.10 shows that finding a counterexample using the conformance oracle is the most difficult for mutants 2 and 5, however, it is nearly trivial for the other four mutants, because for these mutants the final hypothesis only has between 16-20 states. We further investigate mutant 5 by scaling up the **rn** parameter in order to discover the limits of using the conformance oracle setup and to compare its performance to that of a state-of-the-art model-based testing tool.

Table 8.4 shows that for mutant 5 the counterexample does depend on the **rn** parameter: with a higher **rn** value, more ITIMEOUT inputs are required in the counterexample for proving mutant 5 to be different from the reference implementation with the same **rn** value. We investigate mutant 5 by doing both conformance model-based testing and conformance learning using a conformance oracle for increasing **rn** values as shown in Tables 8.11 and 8.12.

		rn09	**rn10**	**rn11**	**rn12**	**rn13**	**rn14**
vr0-1	states	267	295	323	351	379	407
	OQ	42751	50189	54949	63227	68267	77385

Table 8.11: *Finding a counterexample for mutant 5 using a conformance oracle. In fact we are learning the mutant using the reference model as a conformance oracle until we found a counterexample which proves both the learned mutant model and the mutant implementation different from the reference implementation.*

		rn09	**rn10**	**rn11**	**rn12**	**rn13**	**rn14**
vr0-1	avg. IO symbols	54701	109154	141285	475398	497128	1066146
	std.dev.	39385	90617	95454	475627	541333	803093

Table 8.12: *Conformance testing mutant 5 using JTorX.* ***vrX-Y*** *means $m_i \in [X...Y]$, where m_i is a message in* IREQ(m_1, m_2, m_3)*, e.g. vr0-1 allows values 0 and 1 in a message in* IREQ*.* ***rn*** *refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($rn>5$ and $rn\leq 5$) for the* ITIMEOUT *input from state* WA *to* SF *and from* WA *to* SC*.*

Testing using a conformance oracle is really quick: the experiment typically runs within a few seconds. Furthermore, the number of output queries required to build the hypothesis seems to increase only linearly with increasing **rn**. In contrast, as Table 8.12 shows, the number of IO symbols and its related testing time required by JTorX increases much faster. The reason for this difference between the two methods is that for conformance testing the time required depends on the probability of reaching a state that provides a counterexample, while learning using a conformance oracle is fully deterministic and thus is independent of this probability.

8.6 Conclusions and Future Work

We show how to apply active state machine learning methods to a real-world use case from software engineering: conformance testing implementations of the bounded retransmission protocol (BRP). To the best of our knowledge, this use of active learning methods is entirely novel. We demonstrate how to make this application work by combining active learning algorithms (LearnLib and Tomte) with tools from verification (an equivalence checker, CADP) and testing (a model-based test tool, JTorX).

A nice property of the BRP is that it contains two parameter values (the number of retransmissions **rn** and the range of message values **vr**), which can be increased to obtain increasingly complex protocols. This makes it an ideal use case for state machine learning methods because it allows us to discover the limits of their learning capabilities. We investigated these limits on testing the conformance of six mutant implementations with respect to a given reference implementation. All implementations were treated as black-box software systems. The goal of our experiments was to discover how active learning tools can be used to establish the conformance between the mutant and reference implementations. The results of these experiments can be summarized as follows:

- The problem of test selection is a big bottleneck for state-of-the-art active learning tools. Existing model-based testing tools can be used to make this bottleneck less severe.
- Increasing the number of message values **vr** (the alphabet size) increases the time required for test selection as well as the time needed for finding a hypothesis (using output queries).
- Increasing the maximal number of retransmissions **rn** (the length of counterexamples) increases the time required for test selection much more than the time needed for finding a hypothesis.
- Establishing conformance using an equivalence checker and two learned models is very fast. Using only a single learned model and a model-based testing tool is also fast, but can run into problems because test selection takes much longer than equivalence checking.
- Interestingly, there are cases where learning a mutant model (and checking equivalence) is as fast as model-based conformance testing. In general, however, learning a single model and subsequent testing is faster than learning two models.

Furthermore, we noticed in our experiments that the state-of-the-art LearnLib active learning tool quickly runs into trouble when learning one of these mutant implementations. This case was analyzed separately using a probabilistic model checker (MRMC), and based on this analysis we suggested two ways of improving the performance of the active learning method: using a state-of-the-art model-based test tool (TorXakis) for symbolic evaluation of equivalence queries, using a new learning method based on abstraction refinement (Tomte), and introducing a new way of learning based on the novel concept of a conformance oracle. Such a

conformance oracle effectively learns two models at once and uses the (partially) learned models in an equivalence checker to quickly answer equivalence queries asked by either *learner*. When the models are similar, this can greatly reduce the cost of learning.

The concept of a conformance oracle opens up several interesting directions for future work. In particular, since it can also be used as a model-based tester, it would be interesting to further investigate exactly when and why it can be used to establish conformance more quickly than state-of-the-art model-based test tools. Our study already found one such case in mutant 5, where tools based on random testing are troubled by the low probability of reaching a state that leads to a counterexample. A conformance oracle in combination with an active learning tool finds the same counterexample deterministically and in fewer steps. The concept is also closely linked to transfer learning: it can use a previously learned model to speed up the process of learning a new (similar) model. A conformance oracle, however, can transfer these models during the learning process itself, making it an interesting approach for distributed learning settings.

The BRP use case, including the models as well as all the scripts used to link the different tools together, have been made available online for testing and learning by fellow researchers at `http://www.tomte.cs.ru.nl/`.

8.A Mutants of the BRP Sender

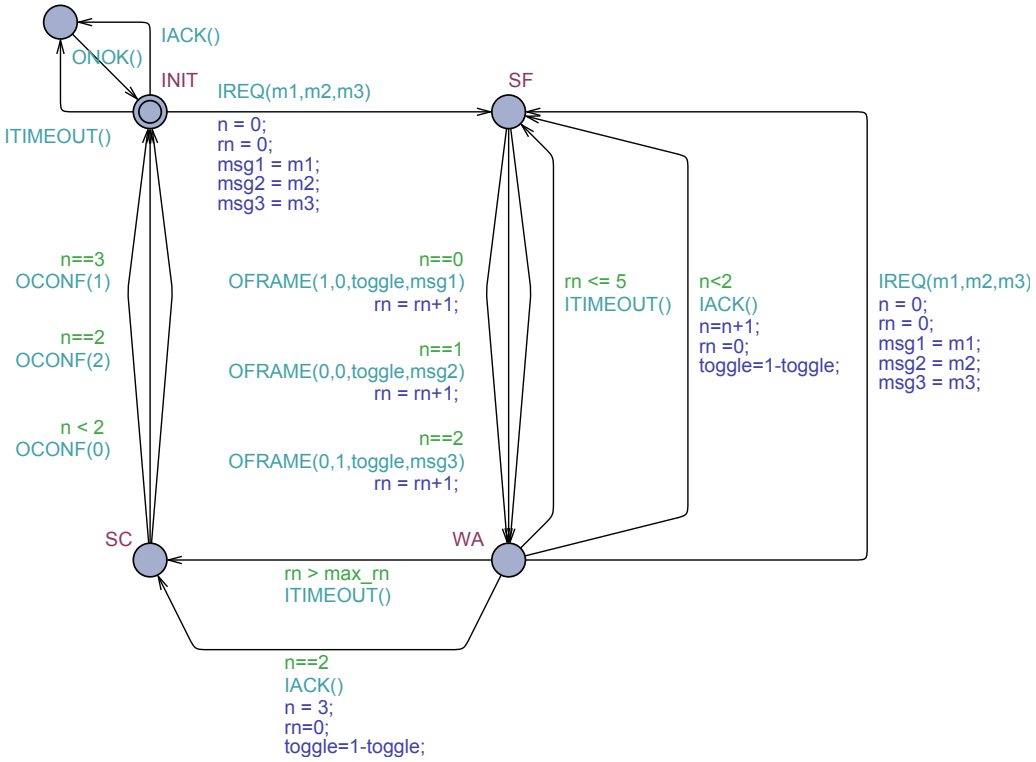


Figure 8.4: *Mutant 1*

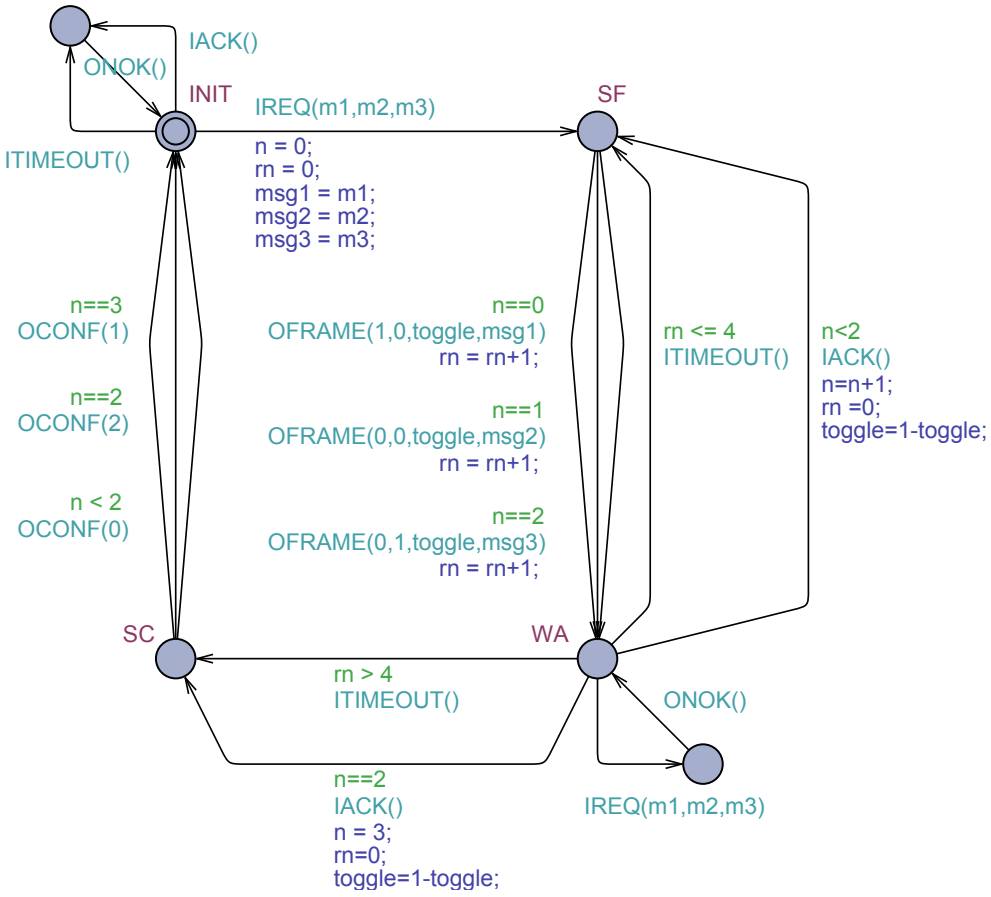


Figure 8.5: *Mutant 2*

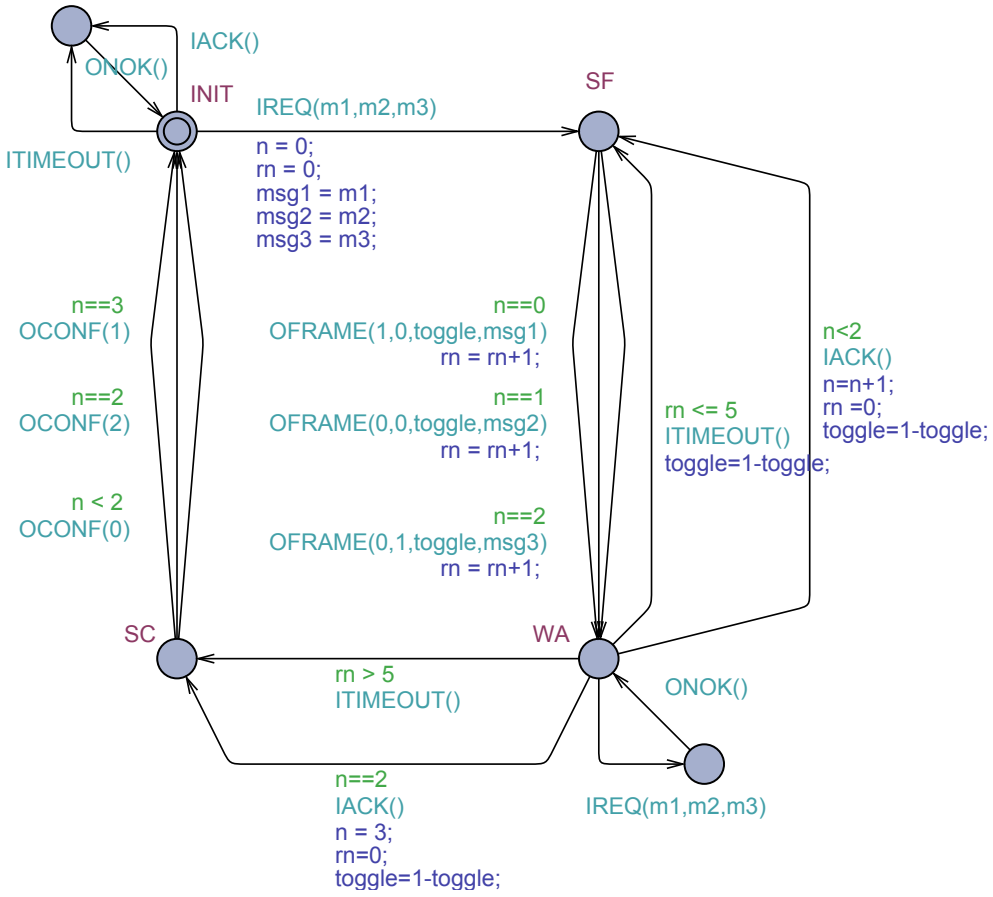


Figure 8.6: *Mutant 3*

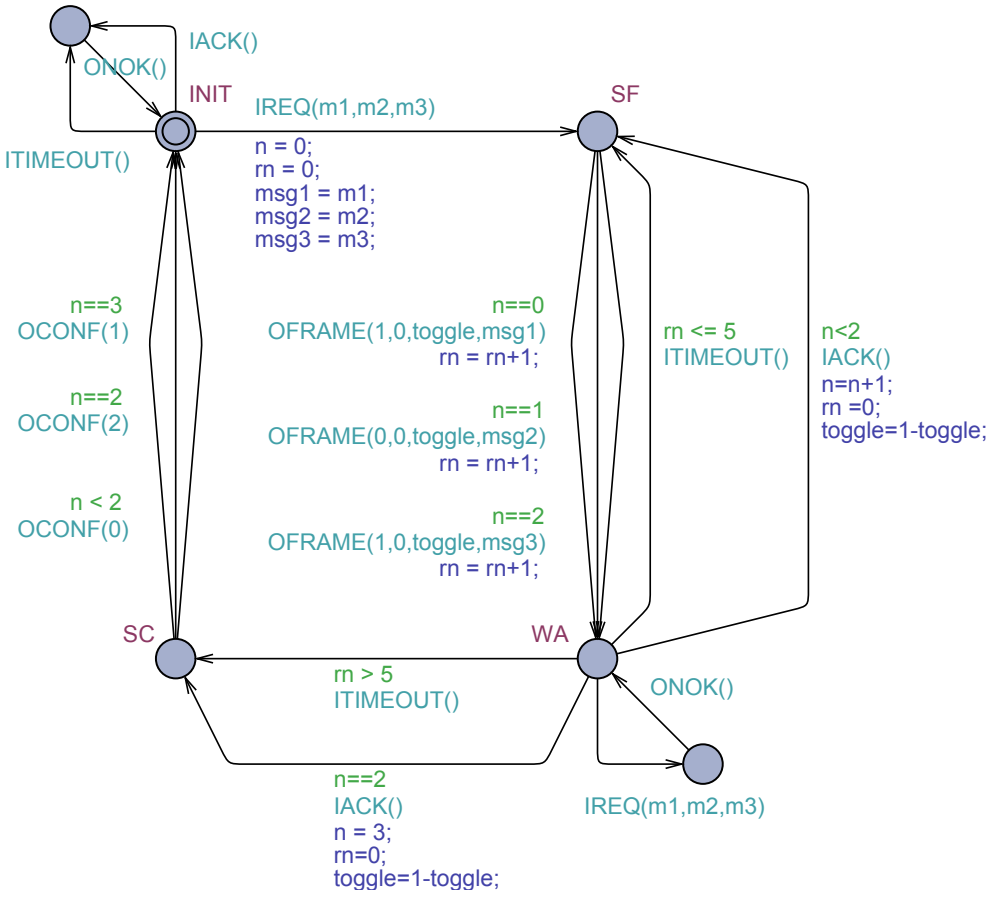


Figure 8.7: *Mutant 4*

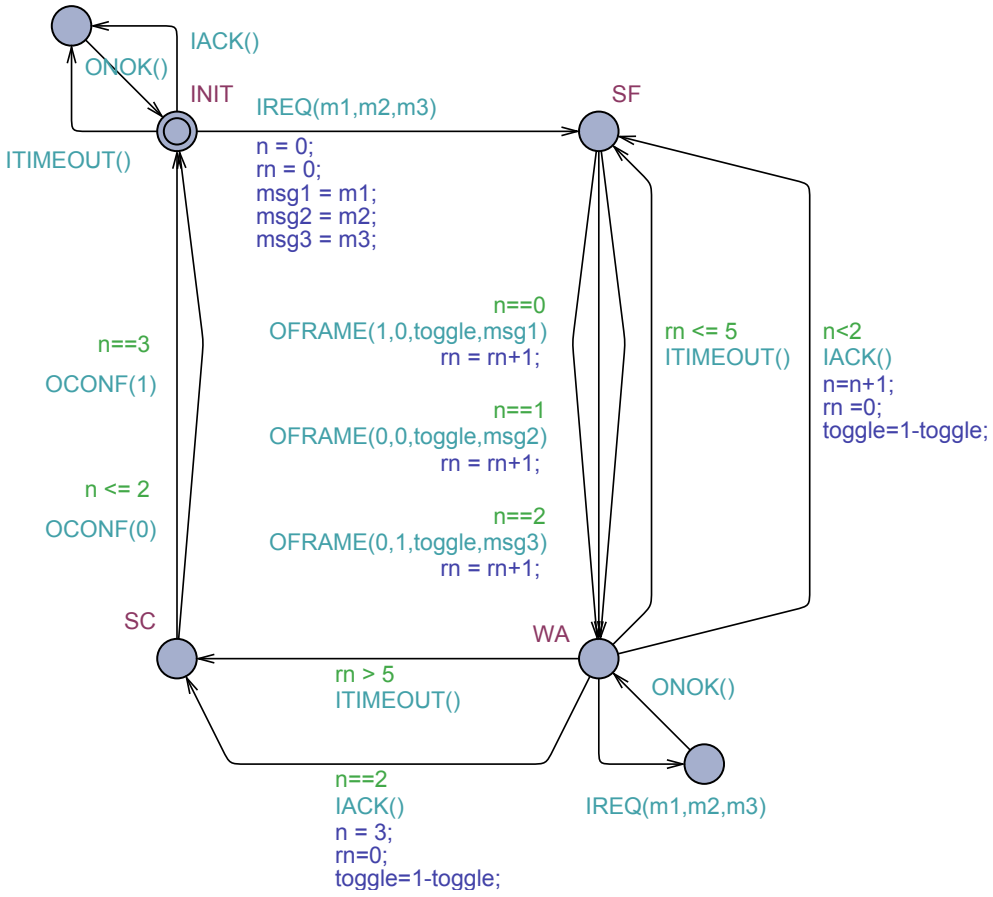


Figure 8.8: *Mutant 5*

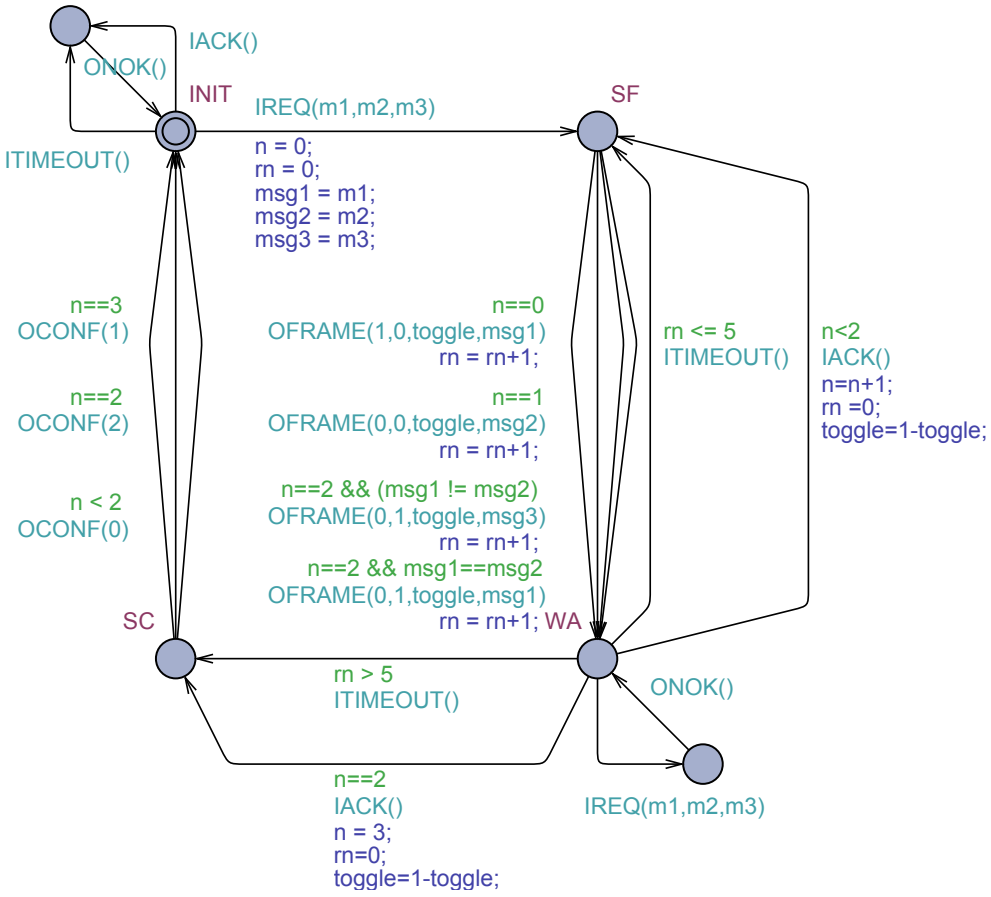


Figure 8.9: *Mutant 6*

8.B Conversions between Input Formats

Figure 8.10 summarizes the various representations of state machines that we use in this chapter, and the conversions between these formats that we have implemented.

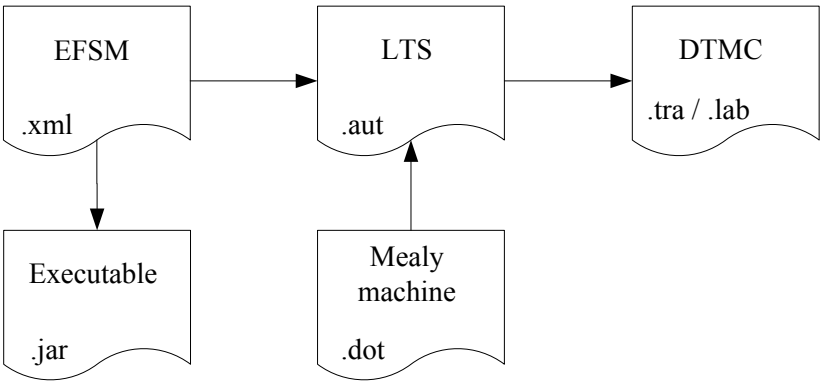


Figure 8.10: *Formats for representing state machines and implemented conversions*

The Mealy machine models learned by LearnLib are represented as `.dot` files. A small script converts Mealy machines in `.dot` format to Labeled Transition Systems in `.aut` format by splitting each transition $q \xrightarrow{i/o} q'$ into a pair of two consecutive transitions $q \xrightarrow{i} q''$ and $q'' \xrightarrow{o} q'$.

Uppaal models, represented as `.xml` files, can be translated to the corresponding implementations, encoded as Java `.jar` files, and to labeled transition systems (LTSs), represented using the `.aut` format.

We use JTorX to establish conformance of mutant implementations to a model of the reference implementation, represented as an `.aut` file.

CADP is used to check strong bisimulation equivalence of labeled transition systems represented as `.aut` files.

Epilogue

Conclusions

In this thesis we have presented an approach to infer models of real-world systems. Standard active learning techniques like Angluin's L^* algorithm typically only perform satisfactory if the state machine has a moderate number of states and actions. Still, practical systems usually have much larger state spaces due to the presence of state variables and data parameters in input messages. We have shown how an intermediate mapper component can be used to bridge the gap between active learning and real-world systems. A mapper maps the extended finite state machine world of real systems into the (Mealy machine) world of active learning algorithms by transforming the large set of inputs of the SUT into a small set of abstract inputs known by the *learner*. We have formalized how the mapper concretizes abstract inputs to concrete inputs and abstracts concrete outputs to abstract outputs. This allows us to reduce the task of the learner to infer an abstract Mealy machine with a small alphabet. We have shown that this abstract model can be turned into a correct EFSM model for the Mealy machine of the implementation by combining the abstract machine with information from the mapper. We have demonstrated the applicability of our approach by manually constructing mappers to infer models of real-world systems like entities in the SIP and TCP protocols, the biometric passport, and the EMV protocol embedded in bank cards.

Furthermore, we have shown that a mapper component can be generated fully automatically. For this purpose we have created an algorithm that automatically performs concretization of abstract inputs and abstraction of concrete outputs without a priori knowledge of the SUT. The algorithm only refines these operations if it turns out that the current abstraction is too coarse and induces nondeterministic behavior. To translate abstract symbols to concrete ones and vice versa, the mapper usually has to be equipped with a number of state variables. We have developed an algorithm that automatically detects new state variables required for learning and extends the mapper component accordingly. We have implemented our algorithms in the Tomte tool and have applied it to several realistic case studies, including the bounded retransmission protocol (BRP) and various components of the alternating bit protocol.

We have investigated the limits of active learning methods during conformance testing of six BRP implementations with respect to a given reference implementation. We have discovered that test selection is a big bottleneck in state-of-the-art active learning tools. Therefore, we have proposed several ways to improve performance of the active learning method. Using a model-based test tool with symbolic test generation like TorXakis allows us to find counterexamples much faster than with tools flattening the data. Similarly, Tomte also exploits the structure of input messages by mapping abstractions to concrete data values, which also facilitates hypothesis verification. As another suggestion, we introduced the new concept of a conformance oracle. Such a conformance oracle enables us to skip the exhaustive task of equivalence approximation by using the reference model as oracle to test equivalence/conformance.

Altogether, we have made a significant step towards applying active learning to real-world systems. We have shown the power of our approach for a wide range of systems.

Future Work

To make automata learning applicable to any real-world system, there are still a number of other limitations and problems that have to be solved.

Extending abstraction techniques The current version of our algorithm is able to learn systems that test for equality of data parameters, but no operations on data are allowed. Therefore, an obvious next step is to extend the abstraction techniques with operations and to allow other comparisons like "less than" or "greater than" in guards. For example a system might react different if the temperature is higher than 40 degrees Celsius. Counters are operations that exist in many systems. If we could extend our abstraction techniques to infer such an operation, we would be able to fully automatically generate models of the TCP and EMV protocol, without suffering from manually constructing a mapper component. Based on the outcomes of previous experiments, the learning algorithm needs to be able to see, for instance, that the second parameter of the third output action is greater than the first parameter of the second input action, or that the last parameter of an output action is a counter that is incremented. One strategy could be that the user specifies the type of operations or guards that should be tested. Another approach could be to use results from the area of machine learning on dynamic detection of likely invariants, in particular the Daikon tool [62]. Daikon is able to spot such relationships between variables.

Learning of nondeterministic systems In Chapter 4 we mentioned that the passport application sometimes exhibits nondeterministic behavior, although the implementation should be deterministic according to the passport specification. In practice, we often encounter systems whose normal behavior is deterministic, but which may exhibit nondeterministic behavior due to exceptions (e.g. a timeout because of long computations or network overload). Therefore, one of the next steps would be to extend our approach to nondeterministic systems. Both in

testing and in learning we might want to drive the SUT to certain states to explore the behavior of the SUT from those states, e.g. in order to execute the same input many times from the same state to check for possible nondeterminism. For this purpose we may for instance reuse the game theory based algorithms of [119, 162] that compute optimal strategies to drive a nondeterministic SUT to certain states while at the same time minimizing the costs of traversal. We could also benefit from existing work on learning nondeterministic systems by Yokomori and Denis et al. [170, 58].

Extending testing techniques As discussed in Chapter 8 test selection and coverage are still a big bottleneck in state-of-the-art active learning tools. In spite of our suggestions for improvement, more work on enhancing testing techniques for equivalence approximation is required. Especially when moving to a nondeterministic setting, current methods are not sufficient any longer. In general, model-based testing is never exhaustive, but for large, nondeterministic systems it is even harder, because we cannot be sure if we have seen all possible behavior for the same input sequence. For example, a system might perform a task successfully in 99% of the cases and return an OK output, but once in a while produce a NOK output. The aim is to measure and quantify this (un)certainty in order to assess the quality of the hypothesized model, as well as to optimize the model-based testing process in order to minimize the probability of accepting an incorrect hypothesized model.

Learning other types of models The theory and approach described in this thesis has been formalized and applied in the setting of Mealy machines. The restriction of Mealy machines that inputs and outputs have to alternate is often inconvenient in practice. We have already shown that any tool for active learning of Mealy machines can be used for learning I/O automata that are deterministic and output determined [11]. We expect that our techniques can also be applied to other more powerful types of models.

Combining inference techniques Combinations with other learning techniques are important in order to choose the best mixture of techniques for obtaining a model for a specific system. For example, one could combine active and passive learning [159, 163, 53] techniques by first constructing a model applying passive learning and then refining it using active learning or by using the passively learned model as an additional oracle during active learning. Other techniques that could be integrated include inference of automata using homing sequences [134] (useful in situations in which the SUT cannot be "reset"), black box checking [125], and the incremental (sequential) learning approach of [111].

Of course the presented as well as new techniques should be applied to more challenging case studies. The experience gained can be used to generate new theory and algorithms, which in turn can be used to further improve the Tomte tool.

Appendix A
List of Symbols

Symbol	Meaning
$\mathcal{A}$	mapper
$\mathcal{E}$	set of edges of an observation tree
$\mathcal{ET}$	set of all event terms
$\mathcal{G}$	set of all formulas over $\mathcal{V}$
$\mathcal{GLT}_{\mathcal{S}}$	set of guard lookahead traces
$\mathcal{H}$	hypothesis (Mealy machine)
$\mathcal{M}$	Mealy machine
$\mathcal{N}$	set of nodes of an observation tree
$\mathcal{O}$	observation table
$\mathcal{OLT}_{\mathcal{S}}$	set of output lookahead traces
$\mathcal{OT}_{\mathcal{S}}$	observation tree
$\mathcal{S}$	scalarset Mealy machine
$\mathcal{SA}$	symbolic mapper
$\mathcal{SM}$	symbolic Mealy Machine
$\mathcal{T}$	set of terms over $\mathcal{V}$
$\mathcal{V}$	universe of variables
$\mathbb{B}$	set of boolean values
$\mathbb{N}$	set of natural numbers
C	set of constants
E	set of event primitives
F	abstraction table
H	set of states of a hypothesis
I	set of (concrete) input symbols
L	set of locations of a scalarset Mealy machine
N	node of an observation tree
N_0	root node of an observation tree
O	set of (concrete) output symbols
P	set of parameters in T_I

Symbol	Meaning
Q	set of states of a Mealy machine
R	set of mapper states
T	set of event terms
U	set of parameters in T_O
V	set of variables
$\mathsf{Val}(V)$	set of all valuations for V
X	set of (abstract) input symbols
Y	set of (abstract) output symbols
a	(input or output) symbol
c	constant
d	parameter value
e	term
f	fresh value
g	guard
h	state of a hypothesis
h_0	initial state of a hypothesis
i	(concrete) input symbol
j, k, m, n	index
l	state of a scalarset Mealy machine
l_0	initial state of a scalarset Mealy machine
lt	lookahead trace
o	(concrete) output symbol
p	parameter
q	state of a Mealy machine
q_0	initial state of a Mealy machine
r	state of a mapper
r_0	initial state of a mapper
s	sequence of output symbols
t	term
u	sequence of input symbols
v	variable
w	sequence of input and output symbols
x	abstract input symbol
y	abstract output symbol
$\alpha_{\mathcal{A}}$	abstraction induced by $\mathcal{A}$
γ	function assigning a value to each constant
$\gamma_{\mathcal{A}}$	concretization induced by $\mathcal{A}$
δ	update function
ϵ	empty sequence
ε	event primitive
ι	function assigning value d_i to parameter p_i
λ	output function

Symbol	Meaning
ξ	valuation
ϱ	update of state variables
$\tau_{\mathcal{A}}$	observation abstraction function induced by $\mathcal{A}$
φ	formula
Γ	set of transitions of a scalarset Mealy machine
Δ	set of symbolic transitions
Θ	initial condition
Σ	event signature
Ψ	set of event abstractions
$\bot$	undefined value
$\models$	guard fulfillment
$[\![\ ]\!]$	semantic evaluation
$\circ$	function composition
$\rightarrow$	transition relation
$\Rightarrow$	transition relation extended to sequences
$\equiv$	syntactic equality (of terms)
$\approx$	observation equivalence (of Mealy machines)
$\leq$	implementation preorder / behavior inclusion (of Mealy machines)
$\approx_{wb}$	observation congruence (of CCS expressions)
abstr	abstraction function
even	function obtaining all even elements of a sequence
$memV(u)$	set of memorable values after u
obs	set of observations
odd	function obtaining all odd elements of a sequence
zip	function zipping input and output sequences together

Appendix B
List of Terms

Bibliography

[1] F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *ICST Workshops*, pages 461–468. IEEE, 2013. Cited on page 13.

[2] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer Berlin Heidelberg, 2012. Cited on page 14.

[3] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample-guided abstraction refinement. Technical report, Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, the Netherlands, 2012. Cited on pages 88 and 91.

[4] F. Aarts, F. Heidarian, and F. Vaandrager. A theory of abstractions for learning interface automata. In M. Koutny and I. Ulidowski, editors, *23rd International Conference on Concurrency Theory (CONCUR), Newcastle upon Tyne, UK, September 3-8, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 240–255. Springer, Sept. 2012. Cited on pages 15 and 43.

[5] F. Aarts, F. Howar, H. Kuppens, and F. Vaandrager. Algorithms for inferring register automata - a comparison of existing approaches. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 6th International Symposium on Leveraging Applications, ISoLA 2014, Corfu, Greece, October 8-11, 2014*, Lecture Notes in Computer Science. Springer, 2014. Cited on pages 14 and 137.

[6] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010. Cited on page 13.

[7] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 2014. Cited on page 13.

[8] F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer. Improving active mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014. Cited on page 14.

[9] F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer. Learning and testing the bounded retransmission protocol. In J. Heinz, C. de la Higuera, and T. Oates, editors, *ICGI*, volume 21 of *JMLR Proceedings*, pages 4–18. JMLR.org, 2012. Cited on pages 15 and 151.

[10] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. 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. Cited on page 13.

[11] F. Aarts and F. Vaandrager. Learning I/O automata. In P. Gastin and F. Laroussinie, editors, *21st International Conference on Concurrency Theory (CONCUR), Paris, France, August 31st - September 3rd, 2010, Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010. Cited on pages 9, 15, 69, and 173.

[12] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 98–109, New York, NY, USA, 2005. ACM. Cited on page 11.

[13] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 4–16. ACM, 2002. Cited on page 9.

[14] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. Cited on pages 5, 20, and 21.

[15] J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. *Reverse Engineering, Working Conference on*, 0:169–178, 2011. Cited on page 9.

[16] Axini B.V.: Axini products web page. `http://www.axini.com/producten/model-based-testing/`. Cited on page 4.

[17] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time markov chains. pages 269–276. Springer, 1996. Cited on page 158.

[18] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. Cited on page 121.

[19] J. Balcázar, J. Díaz, and R. Gavaldà. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997. Cited on page 5.

[20] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. pages 1–3, 2002. Cited on page 4.

[21] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, Jan. 2002. Cited on page 12.

[22] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a stateful network protocol fuzzer. *Information Security*, pages 343–358, 2006. Cited on page 82.

[23] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full–duplex transmission over half–duplex links. *Commun. ACM*, 12:260–261, 1969. Cited on pages 98, 143, and 148.

[24] G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In M. Bernardo and F. 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. Cited on pages 98, 128, and 149.

[25] A. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *TACAS*, pages 266–270, 2010. Cited on pages 6, 61, 62, 67, 145, and 156.

[26] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *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. Cited on page 144.

[27] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin Heidelberg, 2006. Cited on page 10.

[28] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin Heidelberg, 2008. Cited on page 10.

[29] J. Bergstra, A. Ponse, and S. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001. Cited on page 42.

[30] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 141–150. ACM, 2009. Cited on page 9.

[31] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007. Cited on page 12.

[32] D. Beyer and S. Löwe. Explicit-state software model checking based on cegar and interpolation. In V. Cortellessa and D. Varró, editors, *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin Heidelberg, 2013. Cited on page 12.

[33] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: The automata learning framework. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2010. Cited on page 24.

[34] M. Botinčan and D. Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 443–456, New York, NY, USA, 2013. ACM. Cited on page 11.

[35] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005. Cited on page 4.

[36] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04: 26th Int. Conf. on Software Enginering*, May 2004. Cited on page 10.

[37] BSI. Advanced security mechanisms for machine readable travel documents - extended access control (eac) - version 1.11. Technical Report TR-03110, German Federal Office for Information Security (BSI), Bonn, Germany, 2008. Cited on pages 61 and 63.

[38] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In T. Bultan and P.-A. 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 366–380. Springer, 2011. Cited on pages 10 and 57.

[39] J. Castro and R. Gavaldà. Towards feasible PAC-learning of probabilistic deterministic finite automata. In *ICGI*, pages 163–174, 2008. Cited on page 9.

[40] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In E. Al-Shaer, A. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010. Cited on page 11.

[41] T. Chow. Testing software design modeled by finite-state machines. 4(3):178–187, May 1978. Special collection based on COMPSAC. Cited on pages 64, 66, and 74.

[42] A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, pages 473–497, 2004. Cited on page 9.

[43] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 54–56. Springer, 1997. Cited on page 4.

[44] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000. Cited on page 12.

[45] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. Cited on pages 12, 30, and 87.

[46] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Berlin Heidelberg, 2005. Cited on page 12.

[47] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. volume 2619, pages 331–346, 2003. Cited on page 9.

[48] D. Combe, C. de la Higuera, and J.-C. Janodet. Zulu: An interactive learning competition. In *Proceedings of the 8th International Conference on Finite-state Methods and Natural Language Processing*, FSMNLP'09, pages 139–146, Berlin, Heidelberg, 2010. Springer-Verlag. Cited on page 24.

[49] P. Comparetti, G. Wondracek, C. Krügel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125. IEEE Computer Society, 2009. Cited on pages 9 and 82.

[50] Conformiq Inc.: Conformiq Inc. products web page for Conformiq Designer. `http://www.conformiq.com/products/conformiq-designer`. Cited on page 4.

[51] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 14:1–14:14, Berkeley, CA, USA, 2007. USENIX Association. Cited on pages 9 and 82.

[52] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM. Cited on page 6.

[53] V. Dallmeier. *Mining and Checking Object Behavior*. PhD thesis, Universität des Saarlandes, Aug. 2010. Cited on pages 4 and 173.

[54] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 17–24. ACM, 2006. Cited on page 9.

[55] P. R. D'Argenio, J.-P. Katoen, T. C. Ruys, and G. J. Tretmans. The bounded retransmission protocol must be on time! In *Proceedings of the 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of* Lecture Notes in Computer Science, pages 416–431. Springer-Verlag, 1997. Cited on pages 131, 143, and 148.

[56] L. de Alfaro and T. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26 of *Software Engineering Notes*, pages 109–120, New York, Sept. 2001. ACM Press. Cited on page 69.

[57] F. H. Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. PhD thesis, Radboud Universiteit Nijmegen, 2012. Cited on page 91.

[58] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using non deterministic finite automata. In *ICGI*, pages 39–50, 2000. Cited on pages 9 and 173.

[59] E. W. Dijkstra. Notes on Structured Programming. circulated privately, Apr. 1970. Cited on page 145.

[60] S. Drimer, S. Murdoch, and R. Anderson. Optimised to fail: Card readers for online banking. In *Financial Cryptography and Data Security*, volume 5628 of *LNCS*, pages 184–200. Springer, 2009. Cited on pages 74 and 80.

[61] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 1-4, 2008. Available at `emvco.com`. Cited on pages 72 and 83.

[62] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. Cited on pages 87 and 172.

[63] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *Advances in System Testing and Validation Lifecycle (VALID 2011)*, pages 109–114, 2011. Cited on page 71.

[64] L. Frantzen, J. Tretmans, and T. A. C. Willemse. Test generation based on symbolic specifications. In *FATES 2004, number 3395 in LNCS*, pages 1–15. Springer-Verlag, 2005. Cited on page 146.

[65] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In P. Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2011. Cited on pages 144 and 147.

[66] D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In A. Miné and D. Schmidt, editors, *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 248–264. Springer Berlin Heidelberg, 2012. Cited on page 11.

[67] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011. Cited on page 8.

[68] O. Grinchtein. *Learning of timed systems*. PhD thesis, 2008. Cited on page 9.

[69] O. Grinchtein, B. Jonsson, and P. Petterson. Inference of event-recording automata using timed decision trees. In *CONCUR*, volume 4137 of *LNCS*, pages 435–449. Springer, 2006. Cited on page 9.

[70] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. volume 2280, pages 357–370, 2002. Cited on page 9.

[71] R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, volume 5047, pages 216–233, 2008. Cited on pages 9 and 10.

[72] O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008. Cited on page 4.

[73] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. volume 2306, pages 80–95, 2002. Cited on page 9.

[74] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994. Cited on page 158.

[75] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings International Workshop TYPES'93,* Nijmegen, The Netherlands, May 1993, volume 806, pages 127–165, 1994. Cited on pages 131, 143, and 148.

[76] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. pages 58–70, 2002. Cited on page 4.

[77] C. d. l. Higuera and J.-C. Janodet. Inference of omega-languages from prefixes. *Theoretical Computer Science*, 313(2):295–312, 2004. Cited on page 9.

[78] K. Hossen, R. Groz, and J. L. Richier. Security vulnerabilities detection using model inference for applications and security protocols. In *Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 534–536. IEEE, 2011. Cited on page 82.

[79] F. Howar. *Active learning of interface programs.* PhD thesis, Technische Universität Dortmund, June 2012. Cited on page 106.

[80] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer Berlin Heidelberg, 2012. Cited on pages 10, 106, 125, 134, and 140.

[81] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2012. Cited on pages 10 and 134.

[82] F. Howar, B. Steffen, and M. Merten. From ZULU to RERS. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010. Cited on page 24.

[83] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011. Cited on pages 12 and 99.

[84] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. *2008 IEEE International Conference on Network Protocols*, 40(2):114–123, 2008. Cited on page 82.

[85] A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, Tallinn, Estonia, June, 2007*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, 2007. Cited on page 8.

[86] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W. H. Jr. and F. 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. Cited on page 9.

[87] ICAO. Doc 9303 - machine readable travel documents - part 1-2. Technical report, International Civil Aviation Organization, 2006. Sixth edition. Cited on pages 61 and 63.

[88] M. I. Inc. Paypass - m/chip technical specifications. Technical report, September 2005. Version 1.3. Cited on pages 80 and 83.

[89] C. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996. Cited on pages 9, 87, 88, and 91.

[90] ISO/IEC. ISO/IEC 7816: Identification cards – Integrated circuit cards. Cited on page 72.

[91] R. Janssen. *Learning a State Diagram of TCP Using Abstraction.* Bachelor thesis, ICIS, Radboud University Nijmegen, 2013. Cited on pages 12 and 54.

[92] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. Prog. Lang. Syst.*, 16(2):259–303, 1994. Cited on page 45.

[93] B. Jonsson. Learning of automata models extended with data. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer Berlin Heidelberg, 2011. Cited on page 11.

[94] J. Jürjens. Model-based security testing using UMLsec:: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008. Cited on page 82.

[95] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Perform. Eval.*, 68(2):90–104, Feb. 2011. Cited on page 158.

[96] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory.* MIT Press, 1994. Cited on page 5.

[97] P. Koopman, P. Achten, and R. Plasmeijer. Model-based shrinking for state-based testing. In J. McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, pages 107–124. Springer Berlin Heidelberg, 2014. Cited on page 121.

[98] S. Lagerlöf. *The Wonderful Adventures of Nils.* Doubleday, Page, 1917. Cited on page 9.

[99] J. Lancia. Un framework de fuzzing pour cartes à puce: application aux protocoles EMV. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, 2011. Cited on page 82.

[100] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126, 1996. Cited on pages 6, 66, and 145.

[101] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450, 2006. Cited on pages 9 and 10.

[102] X. Li and L. Chen. A survey on methods of automatic protocol reverse engineering. In *Computational Intelligence and Security (CIS 2011)*, pages 685–689. IEEE, 2011. Cited on page 82.

[103] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995. Cited on pages 8 and 30.

[104] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08: 30th Int. Conf. on Software Engineering*, pages 501–510, 2008. Cited on page 10.

[105] O. Maler and I. E. Mens. Learning regular languages over large alphabets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014*, Grenoble, France, April 2014. LNCS. Cited on page 11.

[106] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, COLT '91, pages 128–138, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. Cited on page 5.

[107] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100, Washington, DC, USA, 2004. IEEE Computer Society. Cited on pages 5 and 9.

[108] L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37:486–508, 2011. Cited on page 9.

[109] L. Mariani and M. Pezzï£¡. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007. Cited on page 10.

[110] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. Cited on page 17.

[111] K. Meinke. Cge: A sequential learning algorithm for mealy automata. In J. Sempere and P. García, editors, *Grammatical Inference: Theoretical Results and Applications*, volume 6339 of *Lecture Notes in Computer Science*, pages 148–162. Springer Berlin Heidelberg, 2010. Cited on page 173.

[112] K. Meinke and N. Walkinshaw. Model-based testing and model inference. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 440–443. Springer Berlin Heidelberg, 2012. Cited on pages 14 and 144.

[113] M. Merten, F. Howar, B. Steffen, S. Cassel, and B. Jonsson. Demonstrating learning of register automata. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 466–471. Springer, 2012. Cited on page 57.

[114] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation learnlib. In P. Abdulla and K. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011. Cited on pages 10, 21, 24, 50, 61, 64, 74, 128, and 134.

[115] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. Cited on page 42.

[116] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997. Cited on pages 29 and 32.

[117] W. Mostowski and E. Poll. Electronic passports in a nutshell. Technical Report ICIS–R10004, Radboud University Nijmegen, 2010. Cited on page 62.

[118] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, and R. W. Schreur. Model-based testing of electronic passports. In *FMICS '09: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, pages 207–209, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 6, 61, 62, 67, 68, 69, 144, and 145.

[119] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In G. S. Avrunin and G. Rothermel, editors, *ISSTA*, pages 55–64. ACM, 2004. Cited on page 173.

[120] A. Natarajan and P. Balasubramani. *Theory of Computation*. New Age International Publishers, 2003. Cited on page 17.

[121] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. Cited on page 137.

[122] O. Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003. Cited on pages 5, 9, 21, and 23.

[123] The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`. Cited on page 51.

[124] R. J. P. Fiterau-Brostean and F. Vaandrager. Learning fragments of the tcp network protocol. In *Proceedings 19th International Workshop on Formal Methods for Industrial Critical Systems*, 2014. Cited on page 12.

[125] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, Nov. 2001. Cited on pages 9 and 173.

[126] J. Postel (editor). Transmission Control Protocol - DARPA Internet Program Protocol Specification (RFC 3261), September 1981. Available via `http://www.ietf.org/rfc/rfc793.txt`. Cited on pages 54, 55, 57, and 60.

[127] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. Cited on page 158.

[128] QuviQ AB: QuviQ products web page for testing tools. `http://www.quviq.com/products/`. Cited on page 4.

[129] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer*, 11(4):307–324, 2009. Cited on page 14.

[130] H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press. Cited on page 61.

[131] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. Learnlib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009. Cited on pages 10, 21, 24, 50, 61, 64, 74, and 82.

[132] H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In K. Yorav, editor, *Hardware and Software: Verification and Testing*, volume 4899 of *Lecture Notes in Computer Science*, pages 136–152. Springer Berlin Heidelberg, 2008. Cited on page 14.

[133] Reactive Systems, Inc.: The Reactis product line web page. `http://www.reactive-systems.com/products.msp`. Cited on page 4.

[134] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 411–420, New York, NY, USA, 1989. ACM. Cited on pages 5 and 173.

[135] A. W. Roscoe. A classical mind. chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994. Cited on page 4.

[136] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol (RFC 3261), June 2002. Available via `http://www.ietf.org/rfc/rfc3261.txt`. Cited on page 51.

[137] sepp.med GmbH: sepp.med products web page for MBTsuite. `http://www.seppmed.de/produkte/mbtsuite.html`. Cited on page 4.

[138] M. Shafique and Y. Labiche. A systematic review of model based testing tool support. Technical report, Technical Report SCE-10-04, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 2010. Cited on page 6.

[139] M. Shahbaz and R. Groz. Inferring mealy machines. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer Berlin Heidelberg, 2009. Cited on page 9.

[140] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007. Cited on page 10.

[141] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario*. IEEE Computer Society, 2007. Cited on pages 9 and 10.

[142] R. Singh, D. Giannakopoulou, and C. Pasareanu. Learning component interfaces with may and must abstractions. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 527–542. Springer Berlin Heidelberg, 2010. Cited on page 11.

[143] M. Sipser. *Introduction to the Theory of Computation*, chapter Regular Languages, page 76. PWS Publishing Company, 1996. Cited on page 22.

[144] W. Smeenk. *Applying Automata Learning to Complex Industrial Software*. Master thesis, Radboud University Nijmegen, Sept. 2012. Cited on page 24.

[145] W. Smeenk, D. Jansen, and F. Vaandrager. Applying automata learning to embedded control software. Technical report, 2013. Cited on page 9.

[146] B. Steffen, F. Howar, and M. Isberner. Active automata learning: From dfas to interface programs and beyond. In J. Heinz, C. de la Higuera, and T. Oates, editors, *ICGI*, volume 21 of *JMLR Proceedings*, pages 195–209. JMLR.org, 2012. Cited on page 11.

[147] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Berlin Heidelberg, 2011. Cited on pages 5, 9, 21, 23, and 128.

[148] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley Longman, Inc., 1994. Cited on pages 54, 57, and 60.

[149] T. A. Sudkamp. *Languages and Machines: an introduction to the theory of computer science*. Addison-Wesley, third edition, 2006. Cited on page 5.

[150] J.-P. Szikora and P. Teuwen. Banques en ligne: à la découverte d'EMV-CAP. *MISC (Multi-System & Internet Security Cookbook)*, 56:50–62, 2011. Cited on pages 74 and 80.

[151] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Dec. 1992. Cited on page 51.

[152] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. Cited on pages 61 and 62.

[153] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008. Cited on pages 62, 67, and 145.

[154] J. Tretmans. Model-based testing and some steps towards test-based modelling. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer Berlin Heidelberg, 2011. Cited on page 14.

[155] J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003. Cited on pages 62 and 145.

[156] J. Uijen. *Learning Models of Communication Protocols using Abstraction Techniques*. Master thesis, Radboud University Nijmegen and Uppsala University, Nov. 2009. Cited on page 55.

[157] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. Cited on page 145.

[158] W. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011. Cited on page 9.

[159] W. van der Aalst, V. Rubin, H. Verbeek, B. Dongen, E. Kindler, and C. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010. Cited on pages 4 and 173.

[160] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008. Cited on page 8.

[161] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: algorithms and applications. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 137–150. ACM, 2012. Cited on page 19.

[162] M. Veanes, P. Roy, and C. Campbell. Online testing with reinforcement learning. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 240–253. Springer Berlin Heidelberg, 2006. Cited on page 173.

[163] S. Verwer. *Efficient Identification of Timed Automata: Theory and Practice*. PhD thesis, Delft University of Technology, 2010. Cited on pages 4, 9, and 173.

[164] S. Verwer, M. de Weerdt, and C. Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, pages 1–39, 2011. Cited on page 9.

[165] M. Volpato and J. Tretmans. Active learning of nondeterministic systems from an ioco perspective. In T. Margaria and B. Steffen, editors, *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, ISoLA'14. Springer Berlin Heidelberg, 2014. Cited on page 9.

[166] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont. A framework for the competitive evaluation of model inference techniques. In *Proceedings of the First International Workshop on Model Inference In Testing*, MIIT '10, pages 1–9, New York, NY, USA, 2010. ACM. Cited on page 4.

[167] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In A. Petrenko, A. Simão, and J. Maldonado, editors, *Testing Software and Systems*, volume 6435 of *Lecture Notes in Computer Science*, pages 126–141. Springer Berlin Heidelberg, 2010. Cited on page 144.

[168] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218. IEEE, 2007. Cited on page 9.

[169] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5:641–655, 1983. Cited on page 144.

[170] T. Yokomori. Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence 13*, pages 169–189, 1994. Cited on pages 9 and 173.

Samenvatting

Mensen slagen er vaak in om het gedrag van een apparaat of computerprogramma te leren puur door op knoppen te drukken en het resulterende gedrag te observeren. Vooral kinderen zijn hier goed in en weten precies hoe ze een game computer, iPod of magnetron moeten bedienen zonder ooit een handleiding te hebben geraadpleegd. In dit soort situaties vormen we mentaal een *toestandsdiagram*: we reconstrueren in welke toestanden een apparaat of programma zich kan bevinden en welke toestandsovergangen plaatsvinden als gevolg van welke invoer. Dit proefschrift gaat over de vraag hoe we computers zover kunnen krijgen dat ze zelf, door systematisch knoppen in te drukken en de resulterende uitvoer te observeren, complexe toestandsdiagrammen kunnen leren van apparaten of programma's. Standaard algoritmen slagen er in om toestandsdiagrammen te leren met maximaal 30.000 toestanden. Deze algoritmen zijn niet direct toepasbaar voor het leren van het gedrag van realistische ICT toepassingen, aangezien deze toepassingen beschikken over geheugen en er bij invoer- en uitvoeracties ook vaak sprake is van dataparameters (telefoonnummers bij een mobieltje, de kooktijd in minuten bij een magnetron, enz.). Zelfs wanneer we uitgaan van een simpel apparaat met een geheugen van slechts 450 bytes, dan heeft het resulterende toestandsdiagram potentieel meer dan $256^{450} \approx 10^{1000}$ toestanden. We hebben de standaard technieken verder ontwikkeld en software geprogrammeerd waarmee we – *routinematig en volledig automatisch* – toestandsdiagrammen kunnen leren.

Indien ontwikkelaars van software beschikken over modellen van het gedrag van software componenten, dan stelt dit ze veelal in staat om betere software te schrijven in minder tijd. Modellen kunnen bijvoorbeeld worden gebruikt om een systeem te simuleren voordat het wordt gebouwd, bij besprekingen tussen belanghebbenden, voor het automatisch genereren van software, voor het automatisch genereren van tests en bij hergebruik van software. Bij de ontwikkeling van nieuwe systemen worden er daarom tegenwoordig vaak modellen geconstrueerd, bijvoorbeeld in de taal UML. Het construeren van modellen voor bestaande softwarecomponenten, waarvan veelal geen of geen goede documentatie beschikbaar is, vormt in de praktijk echter een enorm probleem.

In Deel II van dit proefschrift wordt beschreven hoe met behulp van een zogenaamde *mapper* component de basis leertechnieken kunnen worden uitgebreid. De grondgedachte is dat een mapper tussen het standaard leeralgoritme en het systeem dat we willen leren – ook *system under test (SUT)* genoemd – wordt geplaatst. Het leeralgoritme bepaalt de volgorde waarin knoppen worden inge-

drukt of abstracte acties (bijvoorbeeld een actie met twee parameters waarvoor dezelfde waarde gekozen moet worden) worden uitgevoerd. De mapper vertaalt deze naar invoer voor de SUT door concrete waarden voor dataparameters te kiezen. De keuze is afhankelijk van de geschiedenis van eerder gestuurde berichten die door de mapper wordt onthouden. De resulterende uitvoer van de SUT wordt vervolgens ook terug vertaald, waarbij de mapper van concrete waarden van parameters in de uitvoer abstraheert. Op deze manier is het mogelijk een groot aantal concrete overgangen in een echt systeem af te beelden op een beperkt aantal abstracte overgangen in het geleerde toestandsdiagram. In dit tweede deel van het proefschrift wordt de notie van een mapper met de bijbehorende abstractie- en concretisatieoperatoren geformaliseerd. Verder wordt aangetoond dat deze aanpak werkt door handmatig een mapper te definiëren en aan te sluiten op een echt systeem. Op deze wijze zijn wij er in geslaagd om toestandsdiagrammen te leren van enkele veelgebruikte communicatieprotocollen (SIP, TCP en het nieuwe biometrisch paspoort).

Deel III presenteert hoe de handmatig geconstrueerde mappers uit Deel II volledig automatisch kunnen worden gegenereerd voor een bepaalde klasse van systemen, waarin getest kan worden op gelijkheid van dataparameters, maar geen bewerkingen op data zijn toegestaan. Met behulp van *tegenvoorbeeld-gedreven abstractieverfijning* kunnen abstracties, in dit geval gelijkheden tussen dataparameters, worden achterhaald. Indien tijdens het testen van een hypothetisch model een invoer tot een andere uitvoer leidt dan eerder voorspeld, dan is er een tegenvoorbeeld gevonden. Als dit tegenvoorbeeld een nieuwe abstractie bevat, wordt deze automatisch toegevoegd om het probleem op te lossen. Ook de mapper is aangepast om zelf te achterhalen welke waarden van eerder gestuurde berichten onthouden moeten worden. Dit is mogelijk door "toekomstige" berichten uit te voeren en te kijken of een waarde ook in de toekomst van belang is. We hebben ons algoritme in het Tomte tool geïmplementeerd en zijn er in geslaagd van verschillende realistische software-componenten, zoals het biometrisch paspoort en het SIP protocol, geheel automatisch modellen te leren.

Deel IV laat zien hoe divers de toepassingsgebieden van actief leren kunnen zijn. Er wordt aangetoond hoe actief leren gebruikt kan worden om te testen of een implementatie overeenkomt met de referentie-implementatie, d.w.z. een implementatie van de specificatie of standaard van het systeem. Met behulp van een bekende industrïele case study, het bounded retransmission protocol, en een unieke combinatie van software tools op het gebied van modelconstructie (Uppaal), actief leren (LearnLib, Tomte), model-gebaseerd testen (JTorX, TorXakis) en verificatie (CADP, MRMC) zijn wij er in geslaagd verschillende implementaties te leren en fouten daarin op te sporen.

Summary

Humans often manage to learn the behavior of a device or computer program by just pressing buttons and observing the resulting behavior. Especially children are very good in doing this and know exactly how to use a game computer, iPod or microwave oven without ever consulting a manual. In such situations we construct a mental model of a *state diagram*: we determine in which global states a device can be and which state transitions and outputs occur in response to which input. This doctoral thesis deals with the design of algorithms that allow computers to learn complex state diagrams by providing inputs and observing outputs. The state diagrams that can be learned by standard techniques have at most 30.000 states. In contrast, the state diagrams that govern the behavior of computing based systems (defined using dozens of state variables) typically have more than 10^{1000} states. We have further developed the standard techniques and have constructed a tool set that allows us to learn – *routinely and fully automatically* – state diagrams.

Once they have high-level models of the behavior of software components, software engineers can construct better software in less time: behavioral models can be used to simulate a system and reason about it, they allow all stakeholders to participate in the development process and to communicate with each other, they can be used to generate and test implementations, and they facilitate reuse. A key problem in practice, however, is the construction of models for existing software components, for which no or only limited documentation is available.

In Part II of this thesis, we describe how by means of a so-called *mapper* component the basic learning techniques can be enhanced. The main idea is to place a mapper in between the standard learning algorithm and the system we want to learn – also known as *system under test (SUT)*. The learning algorithm determines the sequence of buttons or abstract actions (e.g. an action with two parameters to which the same value has to be assigned) to be executed. These are translated by the mapper to inputs for the SUT by selecting concrete values for the data parameters. The selection is dependent on the history of messages sent earlier, which is maintained by the mapper. The resulting output of the SUT is again translated back by the mapper by abstracting away from concrete output parameter values. In this way it is possible to map a large set of concrete transitions of a real-world system to a small set of abstract transitions of the learned state diagram. In this part, the notion of a mapper and the corresponding operations of abstraction and concretization are formalized. Moreover, it is shown that this approach works by manually defining the mapper and connecting it to the real-

world system. We succeeded in learning models of some realistic communication protocols (TCP, SIP, and the new biometric passport).

Part III presents how the manually constructed mappers of Part II can be generated fully automatically for a restricted class of systems, in which one can test for equality of data parameters, but no operations on data are allowed. By means of *counterexample-guided abstraction refinement* abstractions, i.e. equalities between data parameters can be derived. If during testing of a hypothesized model an input leads to a different output than predicted earlier, we have found a counterexample. If this counterexample contains a new abstraction, it is added automatically to solve the problem. Also the mapper has been adapted to detect itself which values of previously sent messages have to be memorized. By running "future" messages, we can find out whether a value is important in the future. We have implemented our algorithm in the Tomte tool and succeeded in learning models of several realistic software components, including the biometric passport and the SIP protocol fully automatically.

Part IV demonstrates how diverse application areas of active learning can be. It is shown how active learning can be used to test whether an implementation conforms to a reference implementation, i.e. an implementation of the specification or standard of the system. Using a well-known industrial case study, the bounded retransmission protocol, and a unique combination of software tools for model construction (Uppaal), active learning (LearnLib, Tomte), model-based testing (JTorX, TorXakis) and verification (CADP, MRMC) we succeeded in learning models of and revealing errors in several implementations.

Curriculum Vitae

Fides Dorothea Aarts

November 10, 1982:
Born in Nijmegen, the Netherlands.

1993 – 2002:
General qualification for university entrance (allgemeine Hochschulreife),
Johanna-Sebus-Gymnasium Kleve, Germany.

2002 – 2006:
Bachelor of Information and Communication Technology,
Fontys University of Applied Sciences Venlo, the Netherlands.

2009:
Master's thesis,
Uppsala University, Sweden.

2006 – 2009:
Master of Science in Computer Science,
Radboud University Nijmegen, the Netherlands.

2013:
Research intern,
Carnegie Mellon Silicon Valley, CA, USA.

2010 – 2014:
PhD student,
Institute for Computing and Information Sciences,
Radboud University Nijmegen, the Netherlands.

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of

Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and*

Practice. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science,

Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants.* Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking.* Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes.* Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12