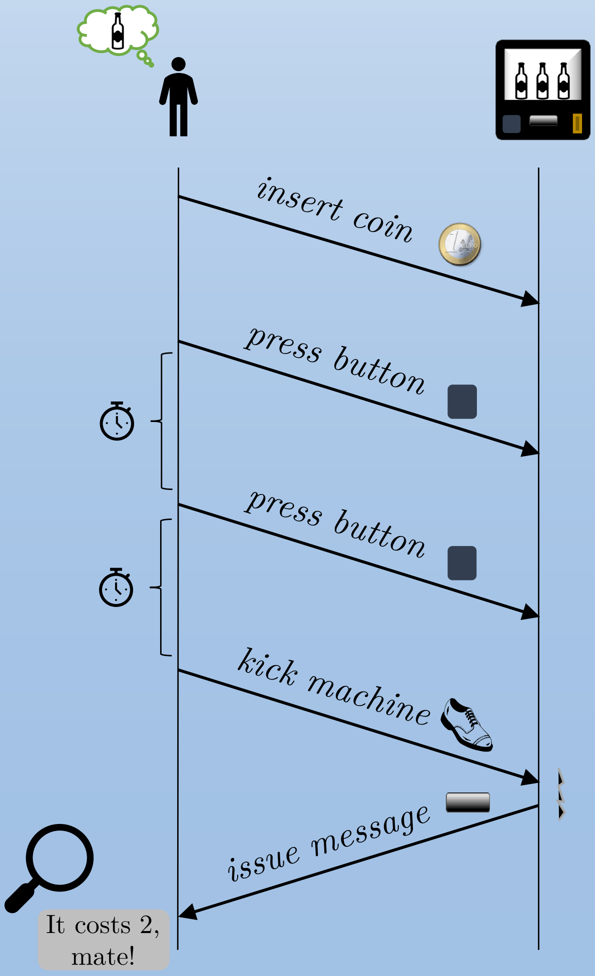
Active Model Learning for the Analysis of Network Protocols



PAUL FITERĂU-BROȘTEAN

Active Model Learning for the Analysis of Network Protocols

Paul Fiterău-Broştean

ISBN: 978-94-028-0963-3
IPA dissertation series: 2018-04

Typeset using LaTeX

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

The work was supported by the NWO (Netherlands Organisation for Scientific Research) project 612.001.216, ALSeP (Active Learning of Security Protocols).

Active Model Learning for the Analysis of Network Protocols

Proefschrift

ter verkrijging van de graad van doctor aan de

Radboud Universiteit Nijmegen op gezag van
de rector magnificus prof.dr. J.H.J.M. van Krieken,

volgens besluit van het college van decanen
in het openbaar te verdedigen op vrijdag 13 april 2018
om 12:30 uur precies

door

Paul Fiterău-Broştean

geboren op 9 augustus 1988
te Timişoara, Roemenië

Acknowledgements

If there is one thing my PhD endeavor has taught me, is that it takes many people to make a thesis. Hence, before diving into the actual content I want to thank the people that were ever so essential in seeing me through this journey.

I first want to thank my supervisor, Frits, for guiding me over the duration of my PhD. Without hesitation, you are the best supervisor I could have hoped for. Your analytical thinking, clear exposition and enthusiasm provided a model for me to follow. Your guidance steered me towards challenging yet interesting and (often) fruitful trails. Your support never waned, even in times of unmet deadlines and broken expectations. Most importantly, I am grateful for the interest you give to the progress of all the students you mentor. I have no doubt that your supervision will see through completion of many future theses.

I would also like to thank the people without whom the PhD journey could not have happened.

Ioana was my supervisor for both my Bachelor's and Master's theses, with the latter providing a bridge to my doctoral research topic. Ioana, I am deeply thankful for your guidance through that early period and for instilling in me the belief that I could actually do a PhD. I was not a model student, yet you were a model supervisor.

My sister, Mădălina, and her husband, Marius, were two people who encouraged me to do an Erasmus exchange semester at Radboud University and, later, to apply for a PhD position at the same university. The exchange semester turned out to be the most exciting semester of my time as a student, while the PhD provided me with a valuable development stage. It is fair to say that I am very grateful to both of them. As I am to Emilia, the professor who approved my last-minute Erasmus application.

Moving on, I would like to thank my research collaborators. I cannot think of one piece of published work where collaboration wasn't essential. Thus, in no particular order, I thank Frits, Ramon, Harco, Falk, Joeri, Erik, Toon, Patrick and Rick. I extend these thanks to members of my thesis committees, that is, Falk, Sicco, Bengt, Joeri, Gergely and Jozef. I appreciate you taking time to read this thesis, give feedback or prepare questions for the defense.

During my PhD, I also had the opportunity to assist with two courses that were very relevant to the research conducted in this thesis. The experience was in itself a journey, one with many lessons learned. I want to thank all people who joined me in this journey. In particular, I thank Gergely and all student assistants, on whom I could always rely.

Last but certainly not least, I would like to thank the people who made this PhD period pleasant.

While working in the ICIS institute, I was lucky to be surrounded by friendly yet also highly capable and motivated colleagues. This ensured not only that my social life was rich, but that I also pushed myself in my research duties. Rick, you were simply put the perfect office mate. Your focus and dedication inspired me in my own work, yet I also enjoyed our many talks on football, food, video games... Rick, Alexis, Joshua, Michele, Nils, Ramon, Petra, Harco, Frits, Jan, you all helped keep the atmosphere spirited with your jokes and interesting stories during the many coffee breaks or learning and testing meetings. Alexis, Joshua and Michele in particular made sure I also had an active night life, that I drink only the best beers and wines, and that I eat very tasty dishes. Also enhancing my social life were Manxia, Marcos, Giso, Markus, Steffen, Jacopo, Henning, Michael, Gabriel and many other ICIS colleagues. Manxia, in particular, became a friend I could always count on for support.

Outside of the work place, I am grateful to my landlords Constance and Joop, who were not only excellent hosts, but also became good friends to me. I will gladly remember our Sundays out to the cinema to see opera recordings. I also thank my choir Campuskoor Veelstemmig. Since joining it I have become happier and oddly, more productive in my research. Then I thank the many other people I have had the pleasure of knowing during my time in the Netherlands, among whom I mention Stijn, Ron, Tessa, Raluca, Denisa and Christos. To these I add friends from back home, who helped me recharge my batteries during my home visits. Thanks Sebastian, Diana, Florin and Sergiu!

Finally, I thank my family for supporting me throughout this journey. My mom, dad, sister, grandparents, uncles, my lovely dog, Peanut, I am all so grateful to have had your trust and support. Thank you!

Paul Fiterău-Broştean

Summary

Network protocols have become deeply ingrained into our everyday lives. Each form of internet communication involves a series of protocols, as do many of the applications we have come to use. These applications include web browsers and servers, mail clients, chat programs, *etc.* Note how these applications communicate with each other to provide a desired service. A web browser for example, communicates with a web server to retrieve a desired web page. It is this very communication that is governed by protocols implemented in the applications' software. Unsurprisingly, verifying that these protocols are properly implemented is of crucial importance, especially when security is considered. Performing such a verification is an arduous task, more so in a black-box setting, where the protocol's source code is not accessible. Let us expand on some of the tasks required by testing techniques commonly used in verification.

Classical testing techniques require the constant manual maintenance of a large test suite, and may fail to spot corner cases where implementations are wrong. Model-based testing uses a model of the protocol to automatically generate tests. Unfortunately, such a model is rarely provided by the protocol's specification, so it needs to be manually constructed and maintained. The technique known as model learning can provide significant relief, as it allows for the automatic generation of models from implementations. The models can then be checked against properties extracted from the protocol specification. This can be done manually by inspecting the models, or automatically via model checking. Either way, the tester's task is greatly facilitated.

One goal of this work is to promote model learning as a viable technique for verifying, or in broader terms, analyzing practical software such as protocol implementations. To that end, in Chapters 2 and 3, we use model learning with abstraction to obtain models of TCP and SSH implementations, respectively. We then perform model checking on these models, in order to analyze the adherence of the learned implementations to the corresponding protocol specifications. This analysis helps uncover various standard violations and bugs.

Another goal is to ease verification of protocol implementations by improving model learning techniques. The challenge posed is that while model learning techniques are

useful, their application to verification is made difficult by the many restrictions they impose on the system we want to verify. Such restrictions may require the system to have parameter-less input/output interfaces, to be deterministic, or to have no temporal dependencies. Overcoming these restrictions may not be possible, or may involve significant manual work. The fewer the restrictions, the easier and wider application of learning techniques becomes. So it is a goal of this work to lift some of these restrictions by expanding or developing new learning techniques.

Many current learning algorithms require the system's behavior to be completely deterministic, and as a result, restrict systems from generating arbitrary, unrelated values in outputs. This restriction greatly limits applicability of learning, as many systems, particularly network protocols, output these values in the form of nonces, sequence numbers and identifiers. Chapter 4 introduces an extension of a well-known framework by which we largely lift this restriction. This extension and other optimizations are integrated in Tomte, the learner implementing this framework, and tested over a series of benchmarks.

Network protocols also commonly perform a wide range of arithmetic operations on data, whereas learning algorithms typically limit these operations to assignments and equality checks. A different learning framework provides means of supporting more advanced operations. RALib, the framework's implementation, supports equality and inequality operations. In Chapter 5, we integrate into RALib extensions for handling inequalities over sums with constants, as well as the extension developed in Chapter 4. Integrating these extensions allows us to infer more detailed models of TCP client implementations. Upon analyzing these models, we find bugs which were made discoverable by the new extensions, and could not have been discovered in our earlier experiments on TCP.

Chapters 4 and 5 shed light on more foundational problems. Active learning algorithms are complex and often tied to the restrictions they impose. This makes them difficult to extend, or to adapt for specific usage scenarios, such as learning a system that cannot be reset. They also require optimization before they can be put to practice due to inefficiencies in the traditional framework. Chapter 6 proposes a learning framework based on SMT for confronting these problems. Within this framework, learning algorithms are expressed by more compact logical formulas. This enables quick prototyping of learning for even advanced formalisms. Breaking away from the traditional framework, our framework also removes the need for optimization and achieves high adaptability. We present extensions of our framework for various formalisms and scenarios. We provide an open-source implementation and use it to assess the framework's effectiveness over a series of benchmarks.

Over the course of this thesis we explore different approaches for learning practical systems. Research on each approach is supported by implementations, experiments or case studies. Future work should evolve these approaches in order to further facilitate and widen their application to practical systems. In doing so, it should make it possible to verify even complex implementations with the simple click of a button.

Samenvatting

Netwerkprotocollen worden steeds belangrijker: bij elke vorm van internetcommunicatie worden diverse protocollen gebruikt. Bij veelgebruikte applicaties, zoals webbrowsers, e-mailclients en chatprogramma's, zien we dat deze applicaties onderling communiceren om een dienst te verlenen. De wijze waarop deze communicatie plaatsvindt, wordt beschreven door protocollen, welke geïmplementeerd zijn in de applicaties. Het is niet verwonderlijk dat het verifiëren of deze protocollen correct geïmplementeerd zijn, erg belangrijk is. Helaas is dit verifiëren een zware taak, vooral in een black-box-omgeving waarin de code niet beschikbaar is. Technieken van model-leren kunnen de taak faciliteren. Deze technieken kunnen worden gebruikt om automatisch modellen voor protocolimplementaties te genereren. Deze modellen kunnen dan handmatig of automatisch gecontroleerd worden op eigenschappen, afgeleid van specificaties zoals RFC's.

Een van de doelen van dit proefschrift is om model-leren toe te passen bij het verifiëren van protocolimplementaties. Hoofdstuk 2 en 3 staan in het teken van dit doel. Hierin gebruiken wij model-leren met abstractie om modellen af te leiden van verschilende implementaties van TCP en SSH. Vervolgens gebruiken we de model-check-technieken om te analyseren of de implementaties zich aan de specificaties houden. Deze analyse hielp bij het vinden van bugs. Echter was abstractie in beide case-studies door handmatige "mapper"-componenten geïmplementeerd. Het maken van deze componenten was tijdrovend.

Een ander doel is om model-leeralgoritmen te verbeteren zodat ze volledig automatisch kunnen worden toegepast op echte systemen. Met dat doel introduceert Hoofdstuk 4 een uitbreiding op een bekend algoritme dat het mogelijk maakt om systemen te leren die willekeurige waarden genereren als uitvoer. De meeste model-leeralgoritmen beperken de operaties die in een systeem zijn toegestaan tot testen van gelijkheid en toekenningen. Er is echter ook een leeralgoritme dat geavanceerde operaties ondersteunt. RALib is een implementatie van dit algoritme. Het ondersteunt zowel testen van gelijkheid als ongelijkheid. In Hoofdstuk 5, voegen wij de ondersteuning

van optellingen met constanten aan RALib toe. Daarna hebben wij RALib gebruikt om getailleerde modellen te genereren voor implementaties van TCP. Analyse van deze modellen bracht twee fouten aan het licht.

Een probleem met de geavenceerde model-leeralgoritmen zoals die van Hoofdstuk 5 is dat zij moeilijk zijn om aan te passen en te optimaliseren. In hoofdstuk 6 geven we een alternatief leerraamwerk, gebaseerd op SMT, om dit probleem aan te pakken. In dit raamwerk kunnen model-leeralgoritmen worden uitgedrukt door compactere logische formules. Dit maakt snelle prototyping van algoritmen met geavanceerde formalismen mogelijk. Bovendien vereist ons raamwerk geen optimisatie. Wij presenteren uitbreidingen van ons raamwerk voor verschillende formalismen en scenario's. Deze uitbreidingen implementeren wij in een open-source tool. We hebben de effectiviteit van ons raamwerk beoordeeld door deze tool te benchmarken.

Dit proefschrift onderzoekt verschillende benaderingen voor het genereren van modellen voor praktische systemen met behulp van model-leren. Deze benaderingen moeten in de toekomst verder ontwikkeld worden. Alleen dan is het mogelijk om systemen te verifiëren door eenvoudig op een knop te drukken.

Contents

Chapter 1

Introduction

People are becoming increasingly reliant on *protocols*. Whether it's drawing money from an ATM or purchasing from a vending machine, our interactions with automated systems follow some form of protocol these systems implement. Indeed, a protocol can be seen as the set of rules which govern our interaction with these systems. These rules define the format and order of messages exchanged with the machines, as well as any action they take on the receiving/sending of each message. The rules forming a protocol are defined in the protocol's specification.

To give an example of such rules, a banking protocol is likely to direct an ATM to only display a balance screen if the user has entered a valid PIN. Similarly, a vending machine's underlying protocol is likely to direct the machine to only issue a product if the user has both pressed a button and inserted a coin. Viewing the insertion of a coin, press of a button and issue of a product, as separate messages we describe in Figure 1.1 normal and abnormal scenarios of interaction between the user and vending machine. By giving the user a free soda, the vending machine deviates from its protocol and costs its maintainers the price of a soda. Failure of systems to correctly implement their respective protocols can lead to problems whose cost far exceeds that of a soda.

The protocols described so far involve people's interaction with machines. Network protocols differ in that the interacting entities are solely hardware or software components. To give an example, let us consider what happens when we access some

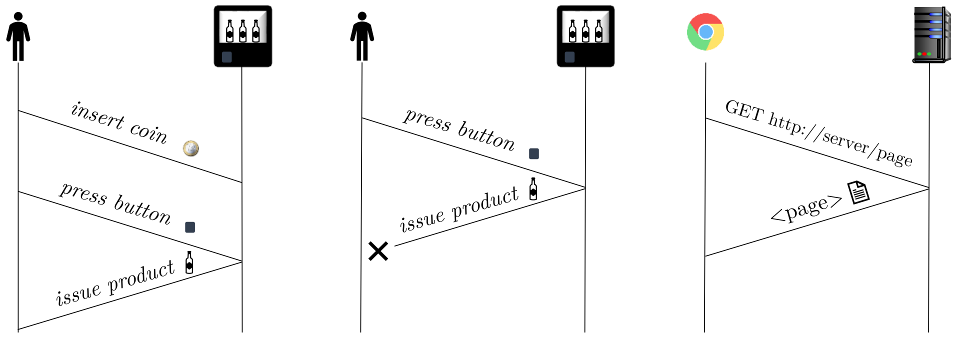


Figure 1.1: Various scenarios of interaction

arbitrary web page. To open the web page in our browser, we first type in the link referring to that web page and press 'Enter'. Our browser then sends a *request message* (or a GET message) to some remote web server asking it for the web page at the given link. The server receives this request, looks up the link in its resource folders, and transmits back a *response message* containing the web page content or an appropriate error. Our browser receives the content or an error, and displays it nicely on screen. The mechanism of requesting and providing these textual resources between clients (our browser) and servers (the remote server contacted) is governed by the HTTP protocol.

This is just one example. The simple transfer of data involved in an HTTP interaction relies on many other protocols. All these protocols are included in the TCP/IP protocol suite, the suite whose protocols govern all forms of interaction over the Internet. Many of these protocols are implemented in our operating systems today. Some are used by popular applications such as Skype, web browsers like Google Chrome or Microsoft Edge, mail clients like Thunderbird or Microsoft Outlook and so on. It is outside the scope of this thesis to delve into the details of this suite, or indeed, into its constituent protocols, hence we refer to [134] for more details.

As in the ATM and vending machine examples, it would be highly problematic if the HTTP implementations of the browser or server would not meet the HTTP protocol's specification. Imagine if our browser occasionally sent malformed requests. The receiving server would then likely reply with error responses even if we supplied valid links, hindering our browsing experience. The consequence of such bugs can be much more severe however. It is only recently that the Heartbleed vulnerability [103] was discovered in a widely used implementation of TLS, the protocol designed to secure traffic over the Internet. Exploiting this vulnerability allowed the theft of passwords and other confidential information. It is said that an estimated 17% of the Internet's secure web servers were vulnerable at the moment of Heartbleed's discovery [104]. More thorough *conformance testing* could have prevented Heartbleed from happening [105]. But what is conformance testing?

Conformance testing is a branch of testing which sees that protocol implementations meet their specifications. It therefore helps reduce the likelihood of these unwanted scenarios. Conformance testing is a form of *black-box* testing, which means it checks the external behavior of an implementation without referring to its internal structure. This contrasts *white-box* testing, which additionally analyzes the program structure. Comparing the two, black-box testing does not require access to the implementation's code, making it more widely applicable. Viewing the system as a black box also has the advantage of decoupling testing from particularities of the actual implementation, such as code structure or the programming language used. This makes testing possible even without prior knowledge of the actual implementation.

There are many ways of performing conformance testing, with each offering a certain degree of *automation*. Automation in turn reduces the work load of the *tester*, the person involved in testing the implementation. Work in this thesis focuses on a conformance testing approach relying on *model learning* which provides a great degree of automation and thus significantly reduces the burden on the tester.

1.1 Ways of Conformance Testing

Checking that implementations *conform* to their specifications is a classical problem. It is hence unsurprising that many approaches have been developed to tackle it. Perhaps the simplest approach one can adopt is to derive a set of *tests* from the specification and run them on the implementation. A test compares the behavior of the implementation to a series of stimulus (or inputs) to the one expected, as according to its specification. Tests initially had to be run manually by the tester. Developments in test methodologies (like keyword driven testing [128]), tools (like Selenium [185]) and frameworks (like JUnit [125]), have made it possible to more easily formulate tests, execute them automatically and to even automatically trace the location of test failure (see [70]). Nevertheless, a problem left unsolved is that a large number of tests have to be manually derived and maintained. This in itself is costly. It is also questionable whether manually written tests cover the corner cases of a specification.

In light of these shortcomings, *model-based testing* [47] has been proposed as a new form of conformance testing which automates both generation and execution of tests based on the *model* of the specification. This model is a formal description of the expected behavior according to the specification. Assuming the model covers all important aspects of the specification, this form of testing can provide high confidence that the implementation, indeed, adheres to its specifications. There are a wide range of tools designed for model-based testing (e.g. Conformiq [71], GraphWalker [97]...). Unfortunately, their application, as is the application of model-based testing in general, is limited by the existence of a suitable model. Specifications are generally textual, even for protocols or other important systems. They do not normally include a formalized model, and if they do, the model generally describes the system at a high level in terms of its normal usage, leaving abnormal usage out. Deriving an adequate model from a specification is therefore far from trivial. And like tests in the previous approach, this model also has to be continuously maintained.

The approach this thesis follows leverages *model learning* [19, 170, 205] to lift the need of providing a model. Model learning is a technique which given a black-box system, can automatically generate (or *infer*) a model corresponding to the system's behavior. This combined with *model checking* [23], a technique enabling the automated checking of models against *formal properties*, should ideally reduce the manual task of a tester to that of formalizing a set of properties from the protocol specification. While protocol specifications don't generally include models, most do formulate a series of requirement statements describing expected behavior (MUST, SHOULD, MAY statements [45]). This lends itself well to the presented approach, as each statement can be encoded in a formal property, which is then verified by performing model checking on the inferred model.

1.2 Research Challenges

A combination of model learning and model checking would appear to provide a near-automated test approach. Unfortunately, challenges in both techniques prevent this appearance from translating to the real world. This work focuses on the learning challenges. That's not to say model checking is free of challenges, on the contrary, state explosion is an active line of research [69]. Currently, however, all that can be learned can be model checked, while the converse is far from true, giving further motivation for our direction.

Perhaps the biggest challenge in model learning comes in the form of the many unrealistic restrictions the technique imposes on the system it is applied to (the system should be simple enough for learning to work). Overcoming these restrictions, if even possible, may require limiting testing to only a subset of the system's functions, or may require a significant amount of manual work, greatly reducing the applicability of model learning. Examples of these restrictions include requiring the system to give the same responses when receiving the same messages (determinism), to be time independent, or implement a *reset* function. It is a challenge of this work to lift or weaken some of these restrictions, thus making model learning more applicable. More specifically, our work tackles restrictions which prevent systems from generating non-deterministic (e.g. random) values in outputs, and also limit their internal operations to equality checks and assignments. Protocols in particular, do generate non-deterministic values and also implement other arithmetic operations.

Another challenge lies in the development of adaptable learning frameworks that can readily support advanced formalisms. Learning algorithms for advanced formalisms are complex and difficult to adapt. They are often tightly bound to the restrictions they impose on the system. Moreover, they can suffer from a blow-up in the number of tests needed due to inefficiencies relating to counterexamples inherent to the classical learning framework. This hinders their applicability to real software. Having this in mind, we propose a framework which, by leveraging SMT solvers, facilitates prototyping of learning for even advanced formalisms. The framework also enables easy adaptations to scenarios in which many other learning approaches are impractical. Such scenarios are when the system cannot be reset, or when only logs of its operation are available. Finally, our framework is free of the inefficiencies affecting the classical framework. Unsurprisingly, benchmarks show it to be competitive with even advanced learning algorithms.

While widening the scope of learning is important, equally important is showing that model learning is a viable strategy for conformance testing (it can help discover bugs or inconsistencies). To that end, we show the usefulness of model learning through case studies on two widely used protocols, TCP and SSH.

1.3 Formalisms for Modeling Software Behavior

As our testing approach involves models, we touch on the formalisms for modeling software behavior featured in this work. With this purpose in mind we define two conceptual client-server protocols, `A` and `B`. We then use variations of *Finite State Machines* (FSMs) to model servers of both protocols. We end by outlining some key characteristics of the models.

Protocol `A` allows a client to connect and send messages to a server. The server ignores any messages prior to a connection. The server acknowledges, by way of acknowledgement messages, the first connection attempt and any messages sent afterward. The server accepts at most one connection and ignores any further connection attempts.

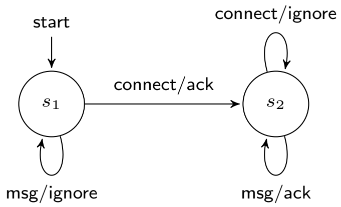


Figure 1.2: Mealy machine model for Protocol `A`

A server for Protocol `A` can be modeled adequately by a *Mealy machine*, as done in Figure 1.2. Mealy machines are FSMs with states and transitions encoding abstract inputs and outputs. Our model has two states (before and after a connection is established), with two abstract inputs (for connecting and sending a message) and two abstract outputs (for ignoring and acknowledging). These messages form the *interface* (or *alphabet*) of our server. To give an interpretation of this model, suppose the server receives a `connect` while in the *initial state* `s1`. The server then transitions to `s2` and generates the output `ack`, acknowledging the client's connection. Once in `s2`, the server stays there, responding to any `msg` inputs by `ack`, and to any `connect` inputs by `ignore`.

Protocol `B` is a refined version of `A`. It enhances the specification with aspects of *data flow*. Both messages and acknowledgements now carry a sequence number. Upon acknowledgement of a connection, an Initial Sequence Number (ISN), which is randomly generated by the server, is communicated to the client. The server then only acknowledges messages with sequence numbers in increasing order, starting from the ISN. Any messages whose sequence numbers fall out of order are ignored. Moreover, acknowledgements sent by the server have sequence numbers equal to those of the messages they acknowledge.

A natural way of modeling servers for Protocol `B` is through *Register Automata* (RA). Figure 1.3a gives an illustration of this. RAs are a variation of *Extended Finite State Machines* (EFSM), and be can seen as expanded versions of Mealy machines. The

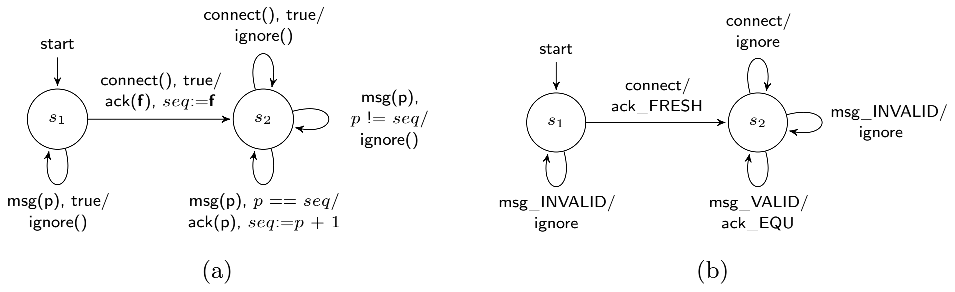


Figure 1.3: Concrete and abstract models for Protocol `B`. **f** as an output parameter means the parameter is assigned a fresh value. In an update **f** refers to the fresh value.

interface of an RA (its inputs and outputs) is parameterized. Moreover, an RA can store parameter values in state variables, and its transitions are extended to contain guards and updates. A transition is fired only if its guard is satisfied, where a guard is a predicate over the input parameters and state variables. Firing a transition additionally executes any updates encoded in it. Outputs may also carry parameters whose value is indicated symbolically by referring to input parameters in the transition, state variables or arbitrary values. We refer to arbitrary values by *fresh values*.

For our example, suppose the server is in state `s1` and receives `connect` from the client. In reaction to the input, the server jumps to `s2`, initializes its state variable *seq* with a fresh value, and communicates this value by packing it in an `ack` output. Thereafter, it only acknowledges `msg` inputs with sequence numbers matching *seq*, and on each acknowledgement it replicates the sequence number of the acknowledged message (i.e. the value of `p` in `msg` is repeated in `ack`, hence the formulation `ack(p)`).

Modeling Protocol `B`'s server as a small Mealy machine is impractical if we consider a large sequence number domain, as Mealy machines cannot model data flow and their interface is abstract. However, using the notion of *abstraction* from [10] we can abstract away from the sequence number parameter found in `msg` and `ack` messages. We do so by remarking that after a client connects, there is only one *valid* sequence number which is acknowledged by the server. All other numbers are *invalid* and ignored, as are numbers in messages prior to a client's connection. Moreover, the first `ack` generated by the server carries a *fresh* value, whereas all other `ack`s carry values *equal* to the valid sequence number. Applying the parameter abstractions corresponding to these remarks (i.e. valid, invalid, fresh, equal), we can model the protocol by the *abstract Mealy machine* of Figure 1.3b. As seen in this example, abstractions provide a way of confining the behavior of the modeled server within the constraints of a Mealy machine.

We end by noting key characteristics of the models (and the servers they describe). Protocol `A`'s model is *deterministic*, that is, for every sequence of inputs the model generates a unique sequence of outputs. By contrast, Protocol `B`'s model is non-deterministic in a concrete sense. This can be evidenced by the model's response to the sequence of inputs `connect`() `msg`(10), which can prompt two different responses depending on the value the model generates upon receiving `connect()` (i.e. if it generates 10 or not). This non-determinism is caused solely by the arbitrary nature in which a fresh value is generated. Abstracting away from this characteristic allows us to produce a deterministic model, such as that in Figure 1.3b. We classify the model in Figure 1.3a as an RA with fresh values. The ability to model fresh values is essential in describing protocols, as many protocols generate them in the form of nonces, identifiers or ISNs, in a similar way to Protocol `B`.

Protocol `B`'s model can also be described in terms of the arithmetics encoded in its guard and updates. In that sense, we can characterize it as an RA with equalities (noting equality and disequality predicates) and successors (noting the successor in one update).

Finally, all models introduced thus far are *transducers*, as they generate an output on every input. All models but the abstract one are also *complete* since their behavior is defined for all inputs in every state. The abstract model is *incomplete* (the output for `msg_VALID` in `s1` is undefined). A different class of models are *acceptors*, which for a sequence of inputs, generate a single boolean output, which is either `true` or `false`. This makes them suitable for describing *languages*, as the output captures inclusion of a sequence of inputs (or *word*) to a language. The output is encoded in the state reached: states can either be accepting or rejecting.

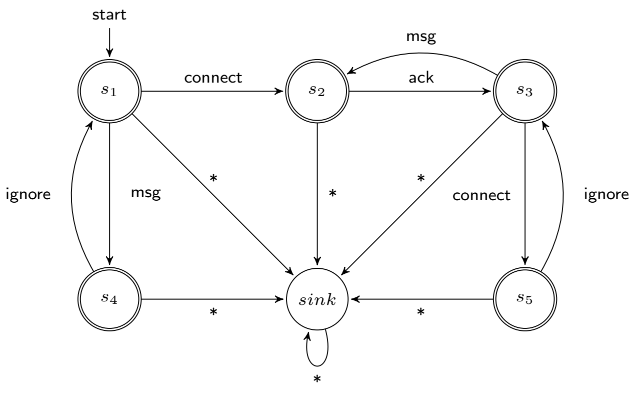


Figure 1.4: Protocol `A` as a DFA. Accepting states have two circles, rejecting states have one.
For each state, '*' indicates all inputs/outputs without an outgoing transition from the state.

Mealy machines are by definition transducers. The literature gives several definitions for describing RA's. Later chapters involving RAs (Chapters 4, 5 and 6) introduce their own RA interpretation. Ultimately, a transducer can be described through an acceptor at the expense of conciseness, by splitting each transition into two separate transitions, connected by an *output state*. We then direct all uncovered transitions to a *sink state*, the acceptor's only rejecting state. Figure 1.4 gives an acceptor representation for Protocol A. The representation is known as a *Deterministic Finite Automaton* (DFA). Modeling transducers as acceptors is done in both Chapters 5 and 6.

1.4 Model Learning Framework

Model learning allows inference of models from black-box systems. There are two settings for model learning, an *active* setting, where inference is done by interaction with a system under learning by exercising the system's interface, and a *passive* setting, where inference is done solely from a set of logs the system generated. Our focus is on an active setting tailored towards the learning of (E)FSMs, like the Mealy machines and RAs described earlier. This setting is also known as *active automata learning*.

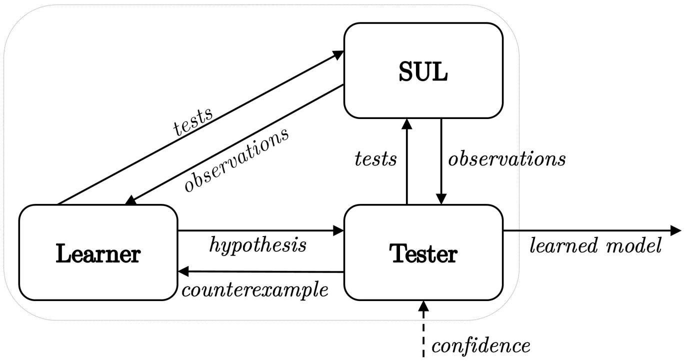


Figure 1.5: Active Model Learning Framework

Figure 1.5 sketches the model learning framework most commonly used in practice. The framework involves three components, the *system under learning* (SUL), the *learner*, a software component implementing a *learning algorithm*, and the *tester*, a software component implementing a (typically model-based) *testing algorithm*.

The learner's goal is to infer a model of the SUL. It does so in an iterative process, by generating and running tests on the SUL. Each test encodes a sequence of inputs. Upon running a test, the learner makes an *observation* of the SUL's response. The learner runs tests until it has made enough observations that it can build a *hypothesis*, a behavioral model consistent with all observations made so far. This hypothesis is sent to the tester, whose task is to check its validity. In case the hypothesis is invalid, the tester may find a *counterexample*. A counterexample is a test which when run on the SUL, results in a different observation than on the hypothesis. The tester gives the counterexample to the learner, which uses it to generate new tests and eventually come up with a *refined hypothesis*. The refined hypothesis is put through the same process, which continues until a hypothesis is judged to be valid by the tester (no counterexample was found), causing learning to terminate and return the hypothesis as the *learned model* of the SUL. The tester can optionally be configured to provide a measurement of confidence in the learned model's correctness, noting that in a black-box setting, correctness can never be guaranteed.

While the SUL is traditionally viewed as a black-box (only its interface is known), the framework can benefit from white-box techniques (that is, techniques which analyze the SUL's internal code). This is especially true with regards to testing, where white-box approaches may ensure coverage of all the code's statements or branches.

1.4.1 A Look Inside the Learner

How can a learner automatically infer a model of the SUL? Learning algorithms are several, yet they all hinge on the discernible nature of states in a model. Concretely, for any two states in a *minimal* model there are *distinguishing sequences* of inputs which, if run from the respective states, will prompt different responses allowing us to tell the states apart. Take Protocol `A` for example, states `s1` and `s2` can be distinguished by the input `connect` which prompts `ack` in `s1` but `ignore` in `s2`. It is important to emphasize the attribute minimal when describing models. Non-minimal models have *equivalent* states which cannot be distinguished. Our active learning setting is concerned with generating minimal models.

Being able to distinguish states allows us to identify them by just accruing for each state, the sequences that differentiate it from the rest. Moreover, it helps in determining a prefix-closed set of *access sequences*, where each access sequence starting from the start state, leads to a different state in the model. For Protocol `A`, the only possible set is $\{\epsilon,\texttt{connect}\}$, where ϵ is the empty sequence. We can verify that this set is prefix-closed: ϵ has no prefixes, whereas `connect` has a single prefix in ϵ, which is included in the set. Having these access sequences and a means of identifying states allows us to construct a model by just checking how the SUL transitions upon receiving each input after each of the access sequences.

These ideas lie at the core of active learning algorithms. Yet how they are implemented varies. Typically learners encode both access sequences and distinguishing sequences in an internal data structure. This data structure is completed with observations made by running tests, until a hypothesis can be generated. Upon processing a counterexample, the learner extracts new access/distinguishing sequences which it uses to update its data structure.

The most common algorithms, including the renowned L^* algorithm, log all tests in an *observation table.* We will use the algorithm described in [198] as reference, though all other table-based algorithms use a similar structure. The rows of an observation table are labeled with *prefixes*, comprising access sequences (we denote by S) and their one-input extensions (or $S \cdot I$, where I is the input alphabet). The columns are labeled with *suffixes*, comprising distinguishing sequences (we denote by D) and singleton sequences for each input in the alphabet. Cells contain the last output generated by running the test formed by concatenating a prefix with a suffix. A state is uniquely determined by the output configuration of a row. Different configurations signal that the respective prefixes lead to different states.

Starting from minimal information (i.e. $\{\epsilon\}$ the set of access sequences and an empty set of distinguishing sequences), new access sequences are added to the table along with their corresponding extensions until the *closedness* condition is met. Under closedness, output configurations of access sequences include output configurations of one-input extensions. This intuitively suggests that all transitions from access sequences lead to already established states. Closedness allows construction of a hypothesis.

We perform a learning run of this algorithm on an adapted version of Protocol `A`. This version following two acknowledged `msg` inputs enters an unresponsive state wherein it ignores all inputs. A model of the adapted protocol is displayed in Figure 1.6. Initially, ϵ is our lone access sequence and we have no distinguishing sequences. We fill in observations for ϵ and its one-input extensions `msg` and `connect`, and obtain Table 1.1. We notice that the table is not closed as there is no access sequence row corresponding to the output configuration of `connect`. This means `connect` leads to a new state, thus should be added to the set of access sequences. We do so, and complete the table again by adding new one-input extension entries, resulting in Table 1.2. This table is closed, which allows us to build a hypothesis resembling Protocol `A`'s model.

		$I \cup D$	
		connect	msg
S	ϵ	ack	ignore
$S \cdot I$	connect	ignore	ack
	msg	ack	ignore

Table 1.1: Table not yet closed. $D = \emptyset$

	connect	msg
ϵ	ack	ignore
connect	ignore	ack
msg	ack	ignore
connect connect	ignore	ack
connect msg	ignore	ack

Table 1.2: First closed table

While this hypothesis is consistent with Protocol `A`, it's inconsistent with the adapted version we are learning. Suppose the tester finds the counterexample `connect msg msg msg` which produces as last output `ignore` on the SUL but `ack` on the hypothesis. A counterexample traverses one or more undiscovered states which cannot be identified using the current set of distinguishing sequences. The learner's goal is to find a non-empty counterexample suffix whose addition to the table makes it unclosed. By doing so the suffix essentially enables the learner to distinguish a new state from existing states and build a new hypothesis. We would also like this suffix to be the shortest, since long suffixes lead to longer, more expensive tests. Hence we iterate the suffixes of the counterexample from shortest to longest until we reach suffix which makes the table unclosed[1]. For our example, this suffix is `msg msg`. The learner adds it to the table as a new distinguishing sequence. The learner then proceeds to run tests in order to close the table again. Table 1.3 encodes the end result. From

[1]Note that this is one of a variation of available strategies for processing counterexamples. Other strategies involve adding multiple suffixes or adding prefixes instead. Also note that using the proposed strategy a counterexample may still be a counterexample for the refined hypothesis.

this, a hypothesis consistent with the adapted protocol is built. Henceforth, no new counterexamples are found and learning finishes.

	con	msg	msg msg
ϵ	ack	ignore	ignore
con	ignore	ack	ack
con msg	ignore	ack	ignore
con msg msg	ignore	ignore	ignore
msg	ack	ignore	ignore
con con	ignore	ack	ack
con msg con	ignore	ack	ignore
con msg msg msg	ignore	ignore	ignore
con msg msg con	ignore	ignore	ignore

Table 1.3: Final table. `connect` appears as `con`

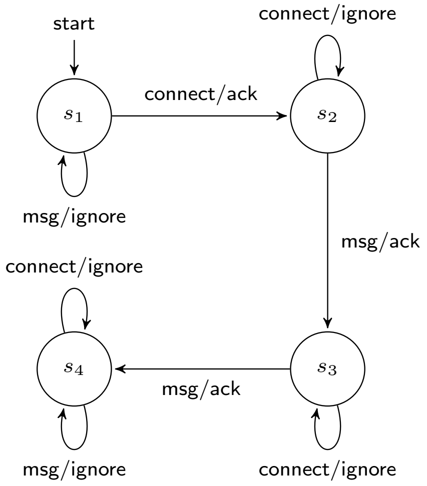


Figure 1.6: Model for adapted Protocol `A`

Having just borne witness to a learning run, it's important to restate that not all learning algorithms use an observation table. Moreover, information encoded in suffixes and prefixes may be different depending on the formalism learned, as will be the information entered in a cell.

1.5 Model Learning Algorithms and Contributions

We shall now go over the learning algorithms and approaches relevant to this work. We use this opportunity to also outline the main contributions.

1.5.1 Classical Learning

The seminal work of Angluin [19] introduced the foundation on which the framework of Section 1.4 is based. It also introduced the L^* learning algorithm, which allows the inference DFAs. DFAs are formalisms used to represent languages. As seen in Figure 1.4, DFAs can also be used to model reactive systems, which display input/output behavior on each transition. Yet they lack conciseness compared to more advanced formalisms, requiring more states to model the same behavior.

This motivated later works [164,187,198] to advance Angluin's algorithm to the setting of Mealy machines. New algorithms for Mealy machines have since been developed with the goal of reducing the number of tests required to infer models. These include the Observation Pack [116] and TTT [122], the later appearing particularly promising.

Testing algorithms for Mealy machines have also been developed [92, 191], some of which provide measurable confidence in the learned models.

1.5.2 Learning with Mappers

Mealy machines provide a conciser way of describing the input/output behavior of real systems, but as shown in Section 1.3, they require abstraction to model parameterized software exhibiting data flow. Work in [10] formally encodes this abstraction in a *mapper* component, which is placed between the SUL and the learner. The mapper confers to the learner an abstract representation of the SUL. An abstract Mealy machine can be inferred, like the one shown for Protocol `B` in Figure 1.3b. By applying the mapper's inverted form on the abstract model we can then generate a concrete model (like an RA). This procedure is also known as *concretization* of the abstract model.

Contribution In Chapters 2 and 3 we use this approach to infer models of real TCP and SSH implementations. We obtain models for various implementations with a measurable degree of confidence attained by applying the testing algorithm in [191]. We then use model checking to verify these models against properties we formalize from the protocols' specifications, and find several standard violations.

The TCP and SSH case studies provide applications of learning with abstraction to widely used protocol implementations. This distinguishes the TCP case study from the work in [10], where the implementation of a TCP simulator was inferred. The two case studies are also among the first to use a combination of model learning and model checking to verify real-world protocol implementations.

In the SSH case study we learn and check SSH server implementations for OpenSSH, BitVise and DropBear. In the TCP case study we learn and check TCP client and server implementations for Windows 8, Linux and FreeBSD. Having obtained models for both clients and servers, we connect them using the model of a loss-less network and check properties concerning their interaction.

1.5.3 Learning with Automatically Generated Mappers

By using mappers an important limitation is lifted, however there is a significant cost incurred. Constructing mappers is an arduous task which often requires deep knowledge of the SUL. Work by Aarts et al. [2] shows that it is possible to construct mappers automatically for a specific class of RAs. The approach hinges on the notion that at any time, only a couple of values (so called *memorable values*) are remembered by the SUL, and are relevant in exploring its future behavior. Parameter abstractions can thus be formed based on relations with these values. Abstractions are refined over the course of learning starting from an initial set of coarse abstractions. As new relations with past values are discovered in counterexamples, new abstractions are encoded into the mapper. This approach falls under the more general approach of Counterexample Generated Abstraction Refinement (CEGAR).

The approach of Aarts et al. was built into Tomte [201], which supports learning RAs with equalities. Tomte's architecture is displayed in Figure 1.7. The Learner implements any active learning algorithm for Mealy machines. The Abstractor encodes the evolving mapper component, tasked with translating between abstract and concrete messages. The Lookahead Oracle is the component responsible with finding memorable values. It passes concrete inputs along to the SUL, while adjoining to each concrete output the set of memorable values in the current run. These are used by the Abstractor to form corresponding output abstractions. Tomte's decoupled nature allows it to easily incorporate and leverage more advanced Mealy machine learners like TTT.

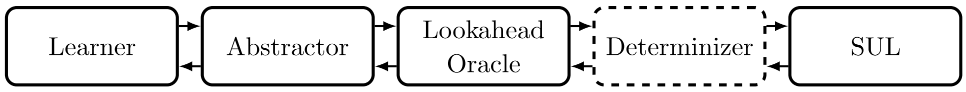


Figure 1.7: Tomte's architecture. The Determinizer is its latest addition

Contribution Chapter 4 extends the work of Aarts et al. to settings allowing fresh values (e.g. randomly generated SUL values). The non-deterministic nature of fresh value generation makes them a problem for learning techniques, which require systems to be deterministic. We combat this by formalizing a *Determinizer* component and incorporating it in Tomte. The Determinizer acts like a mapper, and provides the learner a deterministic concrete view of the SUL, by constructing and applying a 1 to 1 mapping from regular SUL values to 'neat values'. Under the action of the Determinizer, the first fresh value (regardless of its actual value) is mapped to -1, the second fresh value to -2 and so on. We show that learning the behavior of a SUL can be done solely by analyzing its 'neat view'.

The extension to systems with fresh values is essential for the analysis of certain protocols, as explained in Section 1.3. Chapter 4 also introduces a series of optimizations to Tomte. Among the most notable are connecting Tomte to the TTT learning algorithm, and improving and streamlining counterexample analysis. We show gains obtained from these optimizations over an extensive series of benchmarks involving prior configurations of Tomte and RALib.

It should be noted that our approach is not the first to tackle the problem of fresh values. The algorithm introduced by Bollig et al. [37] can also learn models with fresh values, but these models are severely restricted relative to those we can learn. For example, these models cannot describe a language which only accepts sequences of parameterized inputs whose last two values are equal.

1.5.4 Learning Systems with Tree Queries

Cassel et al. [53, 58] presents a different approach for learning RAs which incorporates the handling of parameterized behavior into the learning algorithm. Their algorithm, SL^*, utilizes *tree queries* in place of simple tests. A tree query comprises a concrete prefix and a symbolic suffix. The algorithm poses tree queries to a *tree oracle*, which answers them by generating Symbolic Decision Trees (SDT) describing the SUL's behavior after the prefix for the suffix. An SDT is a data structure which compactly encodes observations made by running a large number of tests on the SUL. Each SDT obtained running a tree query is stored into a tabled structure similar to L^*'s, in the cell corresponding to the query's concrete prefix and symbolic suffix. Closedness checks are done by comparing SDTs in rows for equivalence.

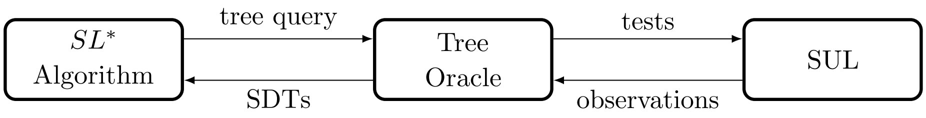


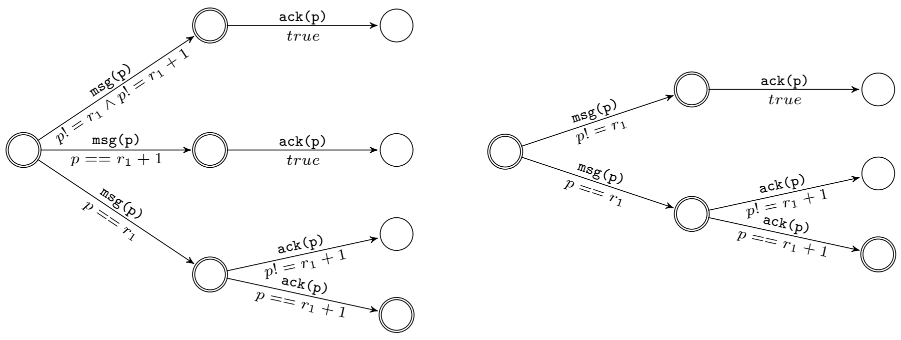
Figure 1.8: RALib's architecture

Canonical implementations of a tree oracle permit the generation of more *succinct* (i.e. compact) models. The framework of Cassel et al. supports learning RAs with advanced relations by providing canonical tree oracles for these relations. In [58], Cassel et. al. formalize tree oracles for equalities and inequalities (involving the $<$,$>$ and $=$ relations). They also give an intuition on how various combinations of relations are handled, including inequalities over sums with constants. They then use a prototype implementation to learn simple models of these combinations. Cassel et al. [54] introduce RALib[2], an open-source implementation of this approach which supports equality and inequality relations.

Before discussing contributions, we give an intuition on the structure of SDTs, and on how a tree oracle can be implemented. Figure 1.9 shows SDTs a tree oracle may construct on a tree query with the concrete prefix `connect ack(10)` and symbolic suffix `msg(p) ack(p)`. An SDT symbolically describes all the instantiations of a suffix that when appended to a prefix, form valid traces of the SUL. These instantiations lead to accepting states in the SDT, whereas those forming invalid traces lead to rejecting states. In the case of Protocol `B`, the suffix forms valid traces if the parameters of `msg` and `ack` are equal to the parameter of `ack` in the prefix and its successor, respectively.

To answer a tree query, a tree oracle as presented in Chapter 5 first generates a *maximally refined tree* which explores all possible parameter configurations for the suffix given the relations. In our example, we consider equality and successor relations. Consequently, we have to explore cases when a suffix parameter is equal, the successor

[2] https://bitbucket.org/learnlib/ralib



(a) Maximally refined SDT (b) Maximally abstract SDT

Figure 1.9: SDTs for prefix `connect ack(10)` and suffix `msg(p) ack(p)`. r_1 refers to the first parameter in the prefix

or different, relative to previous parameters. This requires execution of three tests, which may result in the concrete traces:

`connect() ack(10) msg(10) ack(11)` (equal)

`connect() ack(10) msg(11) nok()` (successor)

`connect() ack(10) msg(20) nok()` (different)

Note, that only the first trace ends in `ack`, and thus matches our suffix. All others don't, hence the cases they encode lead to rejecting states in the tree. For the matching trace, the output value of this `ack` is a successor of the value of the previous `ack`. This automatically invalidates similar traces whose last `ack` contains a value that is not a successor.

Once it has built a maximally refined tree, the oracle compresses it into an equivalent *maximally abstract tree* by merging equivalent subtrees and their respective branches, and returns this tree as answer. This compression step is needed to ensure that learning converges, and also to produce compact models. Notice that the SDT shown in Figure 1.9a is maximally refined only in terms of its input parameters and is already maximally abstract in terms of its output parameters. This was done to ease exposition and also because producing maximally abstract subtrees for output parameters is greatly simplified by the determinism requirement. This requirement means that at most one refined output branch can lead to an accepting state, while all others necessarily lead to rejecting states, allowing for their simple merger.

Contribution In Chapter 5, we extend RALib and use it to generate and check TCP client implementations for FreeBSD and Linux. This is the first practical case study involving an RA learner. We frame the case study within the learning-based testing framework introduced by Meinke [151] (where learning is used as means of building tests more likely to uncover problems). The case study produced detailed concrete models with data that also captured abnormal scenarios. It also lead to the discovery of two new violations. Conducting experiments lead to the uncovering of a bug, whereby while closing a data connection, the Linux TCP client processed and acknowledged certain invalid segments. Uncovering the bug was made possible by the exploratory tests learning involves. The bug was acknowledged and subsequently fixed by developers. Analyzing the models we discover a different violation to the RFC regarding the size of the receive window in TCP, acknowledged by the developers.

Getting RALib to the point where it could learn TCP involved several steps, some of which are detailed in the chapter. First we provide an implementation of the tree oracle for a setting of equalities and inequalities over sums with constants (Protocol `B` would fit in such a setting). We then adapt the Determinizer concept developed in Chapter 4 to this setting, and connect it to the framework. We also implement suffix optimizations for these relations to make the approach more scalable.

The concept of suffix optimization was introduced in [57] for a setting of equalities, but never implemented for our specific setting. This optimization involves annotating the symbolic suffixes obtained from counterexamples, with the relations they capture within the counterexample. The tree oracle only considers these relations when processing tree queries with this suffix, instead of all enabled relations, leading to a reduction in the number of tests needed to answer the tree query. To give a concrete example, in Figure 1.9 knowing that we only have to test the parameter of `msg` for equality (instead of also for successor) would reduce the number of tests from 3 to 2. The reduction becomes (much) more pronounced once we consider more relations, or suffixes and prefixes with more parameters.

1.5.5 An SMT-based Learning Framework

The last two approaches can, in theory, provide automated ways of generating models for many practical systems. However, adapting both approaches to a broader class of systems or learning scenarios is far from trivial. Take for example adaptations for learning systems without resets or learning systems only from a set of logs. Such adaptations would likely mean reconstruction of these approaches from the ground up.

Adding to that, both approaches require a significant number of tests which grows rapidly with increasing system complexity. This was particularly evident in the TCP case study involving RALib, where the high number of tests meant we had to use small input alphabets and could not learn server implementations. Poor scalablity is caused in part by inefficiencies in the classical learning framework which arise when processing counterexamples. Counterexamples driving the learning process often contain complicating information, such as unnecessary inputs or confusing data relations. Unnecessary inputs make counterexamples longer than needed. Confusing data relations make it difficult to identify those which are relevant from a counterexample. To give a concrete example, consider a login system with register and login methods both carrying a user ID and password as parameters. Also consider two counterexample traces exercising the same functionality on the login system: (c1) `register(0,0)` `ok() login(0,0) ok()` and (c2) `register(0,1) ok() login(0,1) ok()`. (c1) contains confusing data relations binding user IDs also to passwords, when in fact, it is irrelevant that they are equal. By contrast, (c2) contains only the relevant relations and is not confusing.

The presence of either unnecessary inputs or confusing data relations in counterexamples can adversely impact the performance of active learning algorithms, causing them to run many more tests (and inputs) than necessary. To give an intuition of the impact, imagine if in the learning run of Section 1.4 we would have found the counterexample `connect msg msg connect connect msg`. Without further processing of this counterexample, we might have very well used the suffix `msg connect connect msg` as a distinguishing sequence. This suffix has twice as many inputs as the compact `msg msg` we used in the learning run, and thus leads to longer tests. The suffix is made longer by two unnecessary `connect` inputs. Confusing data relations hide away the relevant relations. In the context of tree queries, we want to optimize suffix execution only considering relations that are relevant and not those that are irrelevant (such as a user ID being equal to a password).

State-of-the-art algorithms such as TTT effectively tackle the problem of unnecessary inputs for DFAs and Mealy machines. Yet the problem still plagues learners for more advanced formalisms such as RAs. Chapter 4 provides a way of dealing with confusing data relations by a disambiguation step in which all relations are tested, but this procedure is very costly in terms of the number of tests required.

Contribution Chapter 6 proposes a framework based on Satisfiability Modulo Theories (SMT [33]) which intrinsically avoids problems arising with counterexamples. The underlining idea is to separate concerns between the learner and the tester. The learner is no longer able to run tests, its task is reduced to that of generating a hypothesis consistent with a set of observations. The tester is the one performing tests. Counterexamples found by the tester are incorporated by the learner into more refined hypotheses. As it no longer needs to run tests, the learner can also operate in a passive setting, where from a set of logged observations, it can build a hypothesis. By using what is effectively a passive learner in an active setting, we aim to answer a more general question, namely, how does such an approach perform in practical benchmarks compared to the classical active setting using active learning algorithms? As the chapter shows, it is at least competitive.

The proposed framework uses SMT to implement the learner. More specifically, counterexamples found by the tester are encoded into SMT constraints over the functions comprising the formalism definition. The constraints are then supplied to an SMT solver. From the solution provided, the learner generates a hypothesis model which it sends to the tester. This approach benefits from the capacity of SMT solvers to handle advanced arithmetic, which opens the door to the rapid prototyping of learning for advanced formalisms. To that end, we formalize encodings for both conventional FSMs such as DFAs and Mealy machines, and for advanced formalisms such as RAs with equality and fresh values. Our framework is also highly adaptable, as shown in the provided adaptations to learning systems without resets. Additionally, by removing from the learner the ability to run tests, learning performance is no longer affected by complicating information in counterexamples.

We have implemented this framework in the open-source learning tool Z3GI[3], and have shown its effectiveness over a series of experiments, where we compare it to other learners following the classical learning framework. Our tool implements an all purpose learner, in the sense that, it can infer models for many formalisms, including DFAs, Mealy machines, accepting/rejecting RAs and regular input/output RAs (termed IORA in this chapter). It implements learning both actively and passively and can also learn Mealy machines that cannot be reset. Moreover, our tool's decoupled architecture allows encodings to be swapped while the rest of the framework stays the same, facilitating the probing of new encodings.

A setting similar to ours was previously introduced in [213], where the authors connect a passive learner to a model-based tester, though their realization is markedly different, provides no guarantees on the minimality of the learned model and can only learn one specific formalism, in the form of Partial Labeled Transition Systems (PLTS). We additionally compare our approach to the classical one over a series of experiments.

Passive learning using SMT solvers is also not new. Neider et al. [161, 162] propose an SMT-based passive learning approach for FSMs using encodings similar to ours. The approach is shown to be effective even when compared to more involving SAT-based approaches. We improve upon this work adapting the SMT-based approach to richer classes of automata. Moreover, we assess the effectiveness of such an approach when used in an active way, by drawing comparison with classical active learning approaches.

1.6 Related Work

This section gives an overview of works closely related to our area of research. We attempt to group these works, noting that there is a varying degree of interrelation between works of different groups, as there is between active model learning, testing, reverse engineering and other fields.

1.6.1 Applications of Model Learning to Software Analysis

The idea to use model learning as means of analyzing software originates from work by Peled et al. [170], who proposed a conformance testing approach combining model learning via L^* and model checking under the name *black box checking*. The concept was further advanced and applied in [99]. Meinke et al. introduce a similar methodology

[3]https://gitlab.science.ru.nl/rick/z3gi/tree/lata/z3gi

in [149,150] under the name *learning-based testing.* The overriding idea is to use model learning as means of generating better tests, that is, tests that more likely detect non-conformance to supplied specifications. Meinke et al. instantiate the learning-based testing framework, along with specific learning algorithms, for formalisms including first-order functions [149,150] and Kripke structures [151]. Subsequent work introduces the tool LBTest which implements this methodology for Kripke structures [152]. The tool connects the IKL learning algorithm [151] to the NuSMV model checker [165]. Given formal specifications and a black-box system, the tool produces True/False verdicts for compliance to each specification.

Hagerer et al. [101] are the first to frame model learning in a practical setting. They improve the testing of a telecommunications system by leveraging models obtained for its components. Margaria et al. [147] later adapt and apply model learning to derive models of the switch within this system. Model learning has since seen many applications in the area of testing. Raffelt et al. [177] provide a proof of concept for using learning to infer a renown bug tracking system and router [177]. Later work by Windmüller et al. [220] leverages model learning in the regression testing of an editorial system. In a similar case study, Schuts et al. [184] use model learning to improve a new implementation of a Philips legacy control component. Khalili et al. [129] apply model learning to the middleware of a robotic platform. The generated models are then used for the verification of the platform's control software.

Some applications involve security settings. Cho et al. [64] use model learning to infer models of botnets. Analysis of these models uncovered a design flaw in the MegaD botnet's infrastructure. In subsequent work [63], they use the learning tool MACE to analyze implementations of the SMB protocol and were able to find multiple vulnerabilities. De Ruiter et al. [181] generate Mealy machines for different TLS client and server implementation, and discover several implementation flaws which prompted fixes by developers. Tappler at al. [200] infer models of five different broker implementations of the MQTT protocol. They check the models by manually analyzing traces (sequences of inputs/outputs) exposing differences between pairs of the learned models, and find 18 bugs. Other applications of model learning include analysis of a biometric passport [11], several EMV bank cards [3], hand-held readers [59], filter and sanitizer programs [21,22,43], and web applications [24].

1.6.2 Algorithms for Learning Models with Data

Over the course of time, several active learning algorithms have been proposed for inferring models with data. As the algorithms are varied, we describe them more in terms of their restrictions and only occasionally expand upon their inner workings, referring the reader to their corresponding references for details. We also discuss passive learning approaches in closing.

Automata modeling data fit within the broader class of automata, namely, that of automata with (possibly) infinite alphabets. *Finite-memory automata* [126] are among the first formalisms developed within this class. They resemble a restricted class of acceptor RAs with equalities but no message names, which impose restrictions on the way values are stored in registers, such as no two registers can store the same values (we call this *unique-valuedness*). Finite-memory automata are also the first automata with infinite alphabets for which an active learning algorithm was developed. The algorithm of Sakamoto [182] ran iterations of L^* with growing sets of values until the set was large enough to capture the whole language. Each iteration resulted in a DFA, which was subsequently translated to a corresponding finite-memory automata. Scalability of the algorithm is worsened by it having to restart L^* on each counterexample processed. It is still, nevertheless, remarkable that learning for such an expressive class of systems was formulated this early (the work dates back to 1997). Unfortunately, as finite-memory automata were mainly meant as theoretical tools for modeling and proving properties on languages with infinite alphabets, the algorithm has seen limited use in practice.

The first learning algorithm for parameterized systems was introduced by Berg et al. in [30] with the goal of learning protocol entities. The models learned were parameterized, but did not have registers, parameters were restricted to boolean values and only guards were allowed on transitions. Shahbaz and Li et al. [139, 186, 189] remedied this through learning algorithms which supported parameterized systems with unbounded parameter domains. The models inferred still did not have registers, however, and had guards defined over state-local concrete values. Berg et al. in [31] formalize the first algorithm capable of learning fully operational RAs with equalities. Therein, a Mealy machine is first inferred using an alphabet flattened with a small set of values. The machine is then condensed into an RA with equalities. While free of any of the previous restrictions, the approach is hampered by the poor scalability of learning with large alphabets. We can readily note similarities with prior work on learning finite-memory automata.

Earlier attempts on using handcrafted *abstractions* for learning protocol simulations [9], inspired works by Howar and Aarts et al. [7, 117] to formulate automated abstraction-based algorithms for learning RAs with equalities and potentially unbounded parameter domains. These algorithms automatically determine abstractions through iterations of *automated abstraction refinement.* Isberner et al. [120] introduce the idea of *state-local abstractions*, namely, that some abstractions are only relevant in certain states (for example, logging in with a valid username is only relevant in states reached after registering). The idea came in a context where previous approaches employed global abstractions. A new learning algorithm is formulated based on this idea which produces more succint models, while using fewer tests. Abstraction refinement also lies at the core of the algorithm implemented in MACE [63], though that algorithm requires manual specification of an output abstraction function.

Howar et al. [57, 121] introduce an algorithm capable of inferring *canonical* and *succint* acceptor RAs with equalities (or so called *data languages*). Canonicity provides a unique form to all models expressing the same behavior, whereas succintness ensures the capturing of behavior by a compact representation. The canonical form is formulated first in [55] for RAs with equalities, and later extended to RAs with binary relations [56]. The canonical form presented is succint, as it is free of restrictions normally imposed on models with data, like unique-valuedness in the case of finite-memory automata. Some restrictions remain however, notably *right invariance*, which is touched upon in Chapter 4. As an aside, previous works defined similar canonical forms for finite-memory automata in [26,27]. An extension of the RA learning algorithm [58] generalizes it to other operations, while conditioning canonicity to provision of a canonical tree oracle.

The problem of *fresh values* was first tackled by Bollig et al. [37], with an algorithm capable of learning *session automata*. The algorithm builds on the idea that a newly introduced value only has impact within a session. Once the session ends, the value can be forgotten. Session automata are similar to RAs with equalities and unbounded parameter domains, which can express fresh values. Nevertheless, their expressive power is limited. They cannot express, for example, languages containing only sequences of letters where every two consecutive letters are distinct. Fresh values were also tackled in the learner SIMPA [24], which uses a tabled learning algorithm, and *data mining* as means of inferring guards.

More recently, Moerman et al. [156] formulate algorithms based on L^* [19] and its extension to non-deterministic DFAs [36], for learning deterministic and non-deterministic *nominal automata* [35] with equalities. In concise form, these resemble acceptor RAs with equalities without methods, just values (i.e. letters). Their algorithm is based on *nominal set theory* [172]. Prefixes and suffixes are arranged in an observation table as in L^*, but are no longer encoded by single sequences. Instead they are encoded by sets over the infinite alphabet. These sets, while infinite in the number of elements, admit a finite representation which is achieved using *orbits*. For example, supposing a and b are two distinct letters from the alphabet, the orbit of a is the set of all 1 letter sequences, whereas the orbit of ab is the set of all 2 distinct letter sequences. The data structures stored in the table are functions which compactly describe the behavior of the automaton over the set obtained by extending each element of the prefix with each element of the suffix. Their algorithm can be easily adapted to nominal automata with inequalities by virtue of the swappable underlying structure it builds on.

Other works such as [80,146,153] develop algorithms for inferring acceptor *symbolic automata*. These can be seen as acceptor RAs without registers. The algorithms introduced by Botinčan and Argyros et al. [22,43] can learn extended transducer forms of these automata. For example, the algorithm in [43] infers models extended with the notion of a *lookback* parameterized by n, which allows transitions to refer to any of the n last introduced values. Giannakopoulou et al. and Howar et al. [95,157] developed a white-box algorithm to infer *component interfaces* of Java programs. The inferred models resemble symbolic automata; transitions are labeled with method names and guarded with constraints on the corresponding method parameters. The automata capture the safety behavior (i.e. are exceptions generated?) a Java class exhibits by invoking sequences of its public methods.

In the context of passive learning, which entails extracting information from a set of traces, data and control aspects have traditionally been analyzed separately. With regards to data, we mention the tool Daikon, which implements various techniques for invariant extraction from traces generated by execution of an instrumented program [78, 79, 81, 82]. Research on Daikon has spun the development of new techniques that leverage symbolic execution in order to produce fewer and more useful invariants [72, 226]. In the context of control, notable are the classical state machine extraction techniques such as Gold's algorithm, Regular Positive and Negative Inference (RPNI) and Evidence-Driven State Merging (EDSM). These algorithms are used to extract a DFA from a set of positive and negative traces (or samples). We refer the reader to [108] for an overview of these algorithms. More advanced solutions for the same problem have been developed like DFASAT [106], which encodes learning as a Satisfiability (SAT) problem and solves it using a SAT solver. In a similar way, Neider et al. [161, 162] encode learning as an SMT problem and solve it using an SMT solver. Neider's thesis [162] also provides a detailed comparison between the various passive learning approaches.

Data and control are facets that only combined can give a complete description of software behavior. It is only in recent works, that attempts have been made to extract models capturing both aspects. Some approaches extract EFSMs from the specification of a program in the context of *automata-based programming.* Ulyansev et al. [204] use an adaptation of the DFASAT algorithm to infer EFSMs from *test scenarios.* Test scenarios are sequences of elements, where each element comprises the triggering event (can be seen as input), a condition (can be seen as a guard) and one or more outputs generated. Later work [62] extracts automata from both test scenarios and a set of supplied temporal specifications (LTLs). It encodes automata extraction from test scenarios as a Constraint Satisfiability Problem (CSP), which it solves with a CSP solver. Using CSP instead of SAT simplifies the encoding. The EFSM obtained is verified against the supplied LTLs. In case it doesn't conform to all, the model and specification are inputted to an approach based on *ant colony optimization* [61].

Other approaches obtain EFSMs from *program traces* generated from program execution, comprising sequences of method calls joined by valuations over variables and method parameters. These expose less information than test scenarios (guards are hidden) and can be generated in an automated way. Lorenzoli et al. [144] define the GK-tails algorithm which uses Daikon and the K-tails [34] state merging algorithm to obtain EFSMs models from program traces. The EFSMs generated are similar to the RAs presented here. Similar traces are joined together into a single trace. The resulting traces are then decorated using Daikon with invariants over the variables and method parameters. From the decorated traces, a model is generated using K-tails (states whose next K sequences are similar are merged). The model's guards comprise the invariants Daikon derived. Krka et al. [133] adapt the GK-tails algorithm by only allowing merging of states having the same *program state.* Their work evaluates four different strategies for model synthesis based on invariants or trace sequences. In the context of fault analysis, a different approach [148] applies rewriting rules on the traces, replacing concrete data values by abstract ones, and inputs the rewritten traces to a passive FSM learner.

Walkinshaw et al. [215] remarked that while the EFSMs generated by GK-tails are informative and reasonably concise, they are generally non-deterministic (a consequence of transitions being analyzed individually) which reduces their practicality. Moreover, model expressivity is tied to Daikon's invariants, making the approach less flexible. Walkinshaw et al. formulate a different EDSM algorithm which uses data classifiers (such as if-then-else clauses or binary decision trees) to determine which states can be merged, and to implement the transition guards. Data classifiers are generated by applying data classifier inference techniques over the set of traces. They implement this approach in the tool MINT, which they connect to the WEKA [102] classifier framework, providing access to around 100 classifier inference techniques. Approaches so far generate EFSMs that are partial, as they specify only specify guards and not how variables are updated. In a later work, Walkinshaw et al. [214] address this problem by augmenting the EFSMs with update functions inferred by Genetic Programming [132], resulting in fully computational models.

1.6.3 Optimizations to the Active Learning Framework

Recent optimizations of the learning setting have involved reducing learning algorithm overhead, ensuring quality of subsequent hypothesis during learning and better testing. We refer to [119] for a thorough overview of earlier advancements.

Our experience in Chapters 4 and 6 has shown that in terms of the number of tests needed to learn a model, the best performing algorithms are those which require the fewest tests to incorporate counterexamples into their structure and produce new hypotheses. In other words, learners with the least overhead are the most efficient. Recent advancements have tackled this overhead by shorter counterexamples and optimized data structures.

Aarts et al. [2] noted that shorter counterexamples reduce the overall number of tests required, as they result in shorter suffixes. Consequently, they incorporated into their RA learning algorithm, techniques introduced by Koopman et al. [131] for counterexample shrinking. These techniques involve eliminating from counterexamples single transitions or sequences which form loops in the last hypothesis. The effectiveness of applying these techniques was evidenced in [8], which compared existing RA learning approaches. Therein, loop elimination was shown to have a marked effect in shortening counterexamples, and consequently in reducing the number of tests needed for learning.

Reduction in test numbers can also be attained by using optimized data structures. Observation tables provide an intuitive, yet costly way of encoding observations. The cost lies not so much in the memory footprint, as it does in the number of tests needed to close a table and build a new hypothesis. Some of these tests might be meaningless with regards to incorporating the essence of the last counterexample, yet are needed as a side-effect of the structure used and of the redundancies often present in counterexamples. Kearns and Vazirani [127] introduced *discrimination trees* (they call 'classification trees') as an alternative to observation tables. More advanced algorithms such as the Observation Pack [116] and TTT [122] incorporate similar structures. The work in [122] benchmarks different learning algorithms and shows the effectiveness of TTT, and more generally, of algorithms based on discrimination trees

A different line of work follows the quality of hypotheses generated in learning. The aim is to ensure that every new hypothesis is at least just as good qualitatively as the last. Comparing hypothesis is done on the basis of a *distance metric*. Smetsers et al. [197] formalize a metric based on the minimal-length counterexample which distinguishes a hypothesis from the SUL. A hypothesis is better if the minimal-length counterexample is longer. This metric follows the remark of Alfaro et al. [75] that a potential bug in the far-away future is less troubling than a potential bug today.

Smetsers et al. integrate the metric into L^* by adding an additional check performed on each newly generated hypothesis, comparing it to the previous. This comparison results either in a quality guarantee or in a new counterexample for the learner. Later work by Van den Bos et al. [207] enhances Angluin's framework by adding a general *Comparator* component to perform the comparison based on a given metric. They also introduce a new metric centered on the distance of a hypothesis to a set of logs. While ensuring a notion of quality was the main goal, both works note a decrease in the number of tests as a (desirable) side-effect of enforcing these metrics.

Finally, learning correct models cannot be done without effective testers. Our case studies have used the model-based algorithm introduced in [191]. The novelty of the algorithm lies in forming a test by post-pending to a sequence of inputs leading to a state, an adaptive distinguishing sequence which distinguishes this state. By comparison, other model-based algorithms (W-method [65] and Wp-method [92]) post-pend other forms of separating sequences. The conception of this algorithm was prompted by failure of the W and Wp-method algorithms to find counterexamples to an invalid hypothesis in an industrial case study [190]. On the note of separating sequences for states in the model, Smetsers et al. [195] propose a more efficient algorithm for computing them, which can be used to enhance the performance of classical test algorithms like the W-method.

A different approach used in [16] adapts model-based mutation testing (shown to be effective in [14, 15]) for learning Mealy machines. The resulting algorithm compares favorably to that in [191] on the TCP and MQTT models inferred in [88, 200]. Yoo et al. [225] propose a different test approach whereby testing is done using a combination of the W-method and random sampling. Effectiveness is shown through learning experiments on the DNP3 protocol. Alternatively, in a context where several implementations are learned simultaneously, counterexamples can be derived from differences between the hypotheses generated, as done so in [21].

Testing approaches presented so far have been mainly guided by models (model-based). In the context of conformance testing however, it is specifications which we want to check. So it is natural to design tests on the basis of these specifications. To that end, works proposing integration of model checking with model learning [149–152, 170] use counterexamples supplied by the model checker to drive the learning process.

Moving away from black-box settings, white-box methods can also prove effective. For example, the learning algorithms in [63] and [157] use symbolic or concolic execution [130] to instantiate tests exploring paths in programs. The effectiveness of symbolic execution compared to black-box approaches was shown in case studies involving the concolic execution tool JDart [94, 145]. Smetsers et al. [196] propose an alternative approach whereby testing is done by *fuzzing*. Their combined model learning and fuzzing approach scored very well in the RERS 2016 challenge [180], a competition which aims at comparing verification techniques and tools. For fuzzing, they used the tool American Fuzzy Lop (AFL) [13], which helped uncover behaviors that weren't found using an adapted Wp-method.

1.6.4 Tools for Reverse Engineering

Our last section covers tools for reverse engineering. Model learning itself can be viewed as a reverse engineering technique. That said, model learning relies on knowing the SUL's interface. Such knowledge cannot always be guaranteed, especially when considering botnets, whose protocols are purposefully hidden. Consequently, *interface inference* becomes a key problem. Interfaces comprise the message formats an application uses.

Among tools supporting interface inference, Discoverer [73], Biprominer [216], Veritas [218] and ProDecoder [217] generate message formats solely from network traces. To shed insight into how inference can be done, we give a rough description of the mode of operation followed by Veritas and ProDecoder. Raw packets obtained in an observation step are decomposed into sequences of n contiguous bytes (so called n-grams). Keywords are identified from these sequences (for example GET for HTTP), distinguished by the high frequency with which they appear, and then used in machine learning to group messages of the same type into clusters. From each cluster a message format is derived (a way to do this is via sequence alignment techniques such as those in [160] and [83]).

Other tools use the application's binary in *dynamic taint analysis*. This technique involves monitoring how the application processes inputs in order the extract their formats. Polyglot [52], AutoFormat [141], Tupni [74] and Dispatcher [51] fit in this category. We can also include Autogram [111], a tool developed for Java programs (thus uses application bytecode) to learn *context free grammars*.

Having obtained the interface, we can then perform *model inference*. Tools relying on provision of the interface include all active learning tools, of which we mention Tomte[4] and RALib[5] [54] for learning RAs, LearnLib [123][6], AIDE[7] [17] and libalf[8] [140] for learning variations of FSMs. From passive learning tools we mention dfasat[9] [106], StateChum[10] [212] and MINT[11] [215], the latter designed for EFSM synthesis. Other tools integrate interface inference with model inference, Netzob[12] [40,42] being a prime example. Once it has obtained packet formats from traces (using techniques described in [41]), Netzob uses an adaptation of Angluin's active learning algorithm to infer *transition graphs* which can then be used in protocol analysis.

Work in [24] introduces a model-based toolchain for security testing of web applications. Therein, two approaches are described for inferring models of web applications. One relies on a Web crawler to extract an interface and active learning to infer a model using this interface [112], the other uses *static analysis* to generate a model [155].

[4]http://tomte.cs.ru.nl/
[5]https://bitbucket.org/learnlib/ralib/
[6]https://learnlib.de/
[7]http://aide.codeplex.com/
[8]http://libalf.informatik.rwth-aachen.de/
[9]https://bitbucket.org/chrshmmmr/dfasat/
[10]http://statechum.sourceforge.net/
[11]https://bitbucket.org/nwalkinshaw/efsminferencetool/
[12]https://github.com/netzob/netzob/

1.7 Thesis Contents and Personal Contribution

We end the introduction by recapping the contents of the thesis while noting the author's personal contribution. The thesis gathers works from four peer-reviewed articles published at workshops or conferences and one unpublished journal entry which extends a separate article that was also peer-reviewed. Each work is included in a separate chapter. The author has made a significant contribution to each work, nevertheless, it is important to stress out that all works resulted from collaborative efforts.

Chapter 2 describes a case study involving model learning with mappers and model checking TCP implementations. Chapter 2 is based on the following publication:

> P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV 2016*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016 [88]

This publication largely supersedes the following article:

> P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning Fragments of the TCP Network Protocol. In *FMICS 2014*, volume 8718 of *LNCS*, pages 78–93. Springer, 2014 [87]

All data relevant to Chapter 2 has been deposited and is available at:

> P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Source code and data relevant for the paper 'Combining Model Learning and Model Checking to Analyze TCP Implementations'. 2017. doi: 10.17026/dans-xhw-8tyc [86]

Personal Contribution My responsibilities involved implementing the learning setup, formalizing properties from the specifications, running experiments for all implementations, analyzing the learned models and finding inconsistencies. I have also written a considerable part of the article.

Chapter 3 describes a case study involving model learning with mappers and model checking SSH implementations. Chapter 3 is based on the following publication:

> P. Fiterău-Broştean, T. Lenaerts, J. de Ruiter, E. Poll, F. Vaandrager, and P. Verleg. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151. ACM, 2017 [89]

All data relevant to Chapter 3 has been deposited and is available at:

> P. Fiterău-Broştean, E. Poll, F. Vaandrager, T. Lenaerts, J. D. Ruiter, and P. Verleg. Source code and data relevant for the paper 'Model Learning and Model Checking of SSH Implementations'. 2018. doi: 10.17026/dans-z6n-dxq6 [90]

Personal Contribution My responsibilities involved running experiments for BitVise and DropBear, formalizing properties from the specifications and checking them on the learned models. I was the main writer of the article. Moreover, I co-supervised and assisted a Bachelor student with connecting an advanced testing tool to the SSH learning setup, and with learning models for OpenSSH.

Chapter 4 describes an extension of the CEGAR learning algorithm Aarts introduced in [2] to the setting of fresh values. Chapter 4 is based on the following article:

> F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. *Unpublished*, 2016 [6]

This article is due submission at a journal and is an extension of the following publication:

> F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. In *ICTAC 2015*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015 [5]

All data relevant to Chapter 4 has been deposited and is available at:

> F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. Vaandrager. Source code and data relevant for the paper 'Learning Register Automata with Fresh Value Generation'. 2017. doi: 10.17026/dans-zkb-4ppm [4]

Personal Contribution I was a contributor to the idea of a determinizer which spun up during a discussion. I was also involved in implementing the extension and optimizations into Tomte, and running the experiments. In addition, I took part in writing the journal publication, particularly the parts regarding Tomte and the experimental section.

Chapter 5 describes a case study where we used model learning with tree queries to learn TCP client implementations. We then manually checked the obtained models against the specification. Chapter 5 is based on the following publication:

> P. Fiterău-Broştean and F. Howar. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Critical Systems: Formal Methods and Automated Verification*, pages 185–200. Springer, 2017 [84]

All data relevant to Chapter 5 has been deposited and is available at:

> P. Fiterău-Broştean and F. Howar. Source code and data relevant for the paper 'Learning-Based Testing the Sliding Window Behavior of TCP Implementations', 2017. doi: 10.17026/dans-zkt-t8xx [85]

Personal Contribution My responsibilities involved formulating a determinizer for this setting, implementing the necessary extensions into RALib, implementing the learning setup (I was able to re-use some of the components used in Chapter 2), running the experiments, analyzing the models, finding and reporting inconsistencies. I also contributed in writing the publication, particularly the TCP and experimental sections.

Chapter 6 introduces an alternative learning framework and provides an implementation using SMT. Chapter 6 is based on the following publication:

> R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager. Model Learning as a Satisfiability Modulo Theories Problem. *To appear in LATA 2018* [194]

All data relevant to Chapter 6 has been deposited and is available at:

> R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager. Source code and data relevant for the paper 'Model Learning as a Satisfiability Modulo Theories Problem', 2017. doi: 10.17026/dans-xn2-yewe [193]

Personal Contribution Smetsers came up with the idea of the framework, and an initial SMT implementation for FSMs with abstract alphabets. I proposed extending it to RAs and adapting it to systems without resets. I contributed to the formulation of the RA and IORA encodings. I was involved in developing the implementation and conducting experiments. I also contributed in writing, particularly the experimental section.

Chapter 7 draws conclusions and identifies possible lines of future work.

For Chapters 2 through 6 I was also responsible for depositing data relevant to these chapters to the DANS archive[13]. Deposits were made for each chapter. They include source code, models, logs, etc. Also included is documentation which provides a high-level description of the data and gives instructions on how software can be run so as to help reproduce experiments. Readers interested in specific data are referred to these deposits.

[13]https://dans.knaw.nl/en

Chapter 2

Model Learning and Model Checking of TCP Implementations

We combine model learning and model checking in a challenging case study involving Linux, Windows and FreeBSD implementations of TCP. We use model learning to infer models of different software components and then apply model checking to fully explore what may happen when these components (e.g. a Linux client and a Windows server) interact. Our analysis reveals several instances in which TCP implementations do not conform to their RFC specifications.

2.1 Introduction

Our society has become completely dependent on network and security protocols such as TCP/IP, SSH, TLS, Bluetooth, and EMV. Protocol specification or implementation errors may lead to security breaches or even complete network failures, and hence many studies have applied model checking to these protocols in order to find such errors. Since exhaustive model checking of protocol implementations is usually not feasible [124], two alternative approaches have been pursued in the literature. This chapter proposes a third approach.

A first approach, followed in many studies, is to use model checking for analysis of models that have been handcrafted starting from protocol standards. Through this approach many bugs have been detected, see e.g. [28, 48, 109, 135, 142, 199]. However, as observed in [46], the relationships between a handcrafted model of a protocol and the corresponding standard are typically obscure, undermining the reliability and relevance of the obtained verification results. In addition, implementations of protocols frequently do not conform to their specification. Bugs specific to an implementation can never be captured using this way of model checking. In [87], for instance, we showed that both the Windows 8 and Ubuntu 13.10 implementations of TCP violate the standard. In [181], new security flaws were found in three of the TLS implementations that were analyzed, all due to violations of the standard. In [59] and [208] it was shown that implementations of a protocol for Internet banking and of SSH, respectively, violate their specification.

A second approach has been pioneered by Musuvathi and Engler [158]. Using the CMC model checker [159], they model checked the "hardest thing [they] could think of", the Linux kernel's implementation of TCP. Their idea was to run the *entire* Linux kernel as a CMC process. Transitions in the model checker correspond to events like calls from the upper layer, and sending and receiving packets. Each state of the resulting CMC model is around 250 kilobytes. Since CMC cannot exhaustively explore the state space, it focuses on exploring states that are the most different from previously explored states using various heuristics and by exploiting symmetries. Through their analysis, Musuvathi and Engler found four bugs in the Linux TCP implementation. One could argue that, according to textbook definitions of model checking [23, 68], what Musuvathi and Engler do is not model checking but rather a smart form of testing.

The approach we explore in this chapter uses model learning. Model learning, or active automata learning [2, 19, 198], is emerging as a highly effective technique to obtain models of protocol implementations. In fact, all the standard violations reported in [59, 87, 181, 208] have been discovered (or reconfirmed) with the help of model learning. The goal of model learning is to obtain a state model of a black-box system by providing inputs to and observing outputs. This approach makes it possible to obtain models that fully correspond to the observed behavior of the implementation. Since the models are derived from a finite number of observations, we can (without additional assumptions) never be sure that they are correct: even when a model is consistent with all the observations up until today, we cannot exclude the possibility that there will be a discrepancy tomorrow. Nevertheless, through application of conformance testing algorithms [137], we may increase confidence in the correctness of the learned models. In many recent studies, state-of-the-art tools such as LearnLib [198] routinely succeeded to learn correct models efficiently. In the absence of a tractable white-box model of a protocol implementation, a learned model is often an excellent alternative that may be obtained at relatively low cost.

The main contribution of this chapter is the combined application of model checking, model learning and abstraction techniques in a challenging case study involving Linux, Windows and FreeBSD implementations of TCP. Using model learning and abstraction we infer models of different software components and then apply model checking to explore what may happen when these components (e.g. a Linux client and a Windows server) interact.

The idea to combine model checking and model learning was pioneered in [170], under the name of *black box checking*. In [151], a similar methodology was introduced to use learning and model checking to obtain a strong model-based testing approach. Following [151, 170], model checkers are commonly used to analyze models obtained via automata learning. However, most of these applications only consider specifications of a single system component, and do not analyze networks of learned models. An exception is the work of Shahbaz and Groz [188] on integration testing, in which learned component models are composed and then analyzed using reachability analysis in order to find integration faults. Our results considerably extend our previous work on learning fragments of TCP [87] since we have (1) added inputs corresponding to calls from the upper layer, (2) added transmission of data, (3) inferred models of TCP clients in addition to servers, and (4) learned models for FreeBSD in addition to Windows and Linux. Abstraction is the key for scaling existing automata learning methods to realistic applications. In order to obtain tractable models we use the theory of abstractions from [10], which in turn is inspired by earlier work on predicate abstraction [67, 143]. Our use of abstractions is similar to that of Cho et al [64], who used abstractions to infer models of realistic botnet command and control protocols. Whereas in our previous studies on model learning the abstractions were implemented by ad-hoc Java programs, we now define them in a more systematic manner. We provide a language for defining abstractions, and from this definition we automatically generate mapper components for learning and model checking.

Our method may be viewed as a smart black-box testing approach that combines the strengths of model learning and model checking. The main advantage of our method compared to approaches in which models are handcrafted based on specifications is that we analyze the "real thing" and may find "bugs" in implementations. In fact, our analysis revealed several instances in which TCP implementations do not conform to the standard. Compared to the white-box approach of Musuvathi and Engler [158], our black-box method has several advantages. First of all, we obtain explicit component models that can be fully explored using model checking. Also, our method appears to be easier to apply and is more flexible. For instance, once we had learned a model of the Linux implementation it took just two days to learn a model of the Windows implementation. In the approach of [158], one first would need to get access to the proprietary code from Microsoft, and then start more or less from scratch from an entirely different code base. In contrast, using our approach it is possible to learn a model of any TCP implementation within a few days. Besides these practical benefits, there is also an important philosophical advantage. If one constructs a model of some real-world phenomenon or system and makes claims based on this model then, in line with Popper [175], we think this model ought to be falsifiable. Our model of the Windows 8 TCP client is included in the chapter in Figure 2.2, and all Mealy machine and nuSMV models are available at [1]. Our notion of state is clear and based on the Nerode congruence [163]: two traces lead to the same state unless there is a distinguishing suffix. Any researcher can study our models and point out mistakes. In contrast, the model of Musuvathi is specified implicitly through heuristics (when have we seen a state before?) that are programmed on top of the Linux implementation. As a result, falsification of their model is virtually impossible.

[1]http://www.sws.cs.ru.nl/publications/papers/fvaan/FJV2016/

2.2 Background on Model Learning

Mealy machines

During model learning, we represent protocol entities as Mealy machines. A *Mealy machine* is a tuple $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$, where I, O, and Q are finite sets of *input actions*, *output actions*, 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 $(q, i, o, q') \in \rightarrow$. We assume $\mathcal{M}$ to be *input enabled* (or *completely specified*) in the sense that, for each state q and input i, there is a transition $q \xrightarrow{i/o} q'$, for some o and q'. We call $\mathcal{M}$ *deterministic* if for each state q and input i there is exactly one output o and one state q' such that $q \xrightarrow{i/o} q'$. We call $\mathcal{M}$ *weakly deterministic* if for each state q, input i and output o there is exactly one state q' with $q \xrightarrow{i/o} q'$.

Let $\sigma = i_1 \cdots i_n \in I^*$ and $\rho = o_1 \cdots o_n \in O^*$. Then ρ is an *observation* triggered by σ in $\mathcal{M}$, notation $\rho \in A_{\mathcal{M}}(\sigma)$, if there are $q_0 \cdots q_n \in Q^*$ such that $q_0 = q^0$ and $q_{j-1} \xrightarrow{i_j/o_j} q_j$, for all j with $0 \leq j < n$. If $\mathcal{M}$ and $\mathcal{M}'$ are Mealy machines with the same inputs I and outputs O, then we write $\mathcal{M} \leq \mathcal{M}'$ if, for each $\sigma \in I^*$, $A_{\mathcal{M}}(\sigma) \subseteq A_{\mathcal{M}'}(\sigma)$. We say that $\mathcal{M}$ and $\mathcal{M}'$ are *(behaviorally) equivalent*, notation $\mathcal{M} \approx \mathcal{M}'$, if both $\mathcal{M} \leq \mathcal{M}'$ and $\mathcal{M}' \leq \mathcal{M}$.

If $\mathcal{M}$ is deterministic, then $A_{\mathcal{M}}(\sigma)$ is a singleton set for each input sequence σ. In this case, $\mathcal{M}$ can equivalently be represented as a structure $\langle I, O, Q, q^0, \delta, \lambda \rangle$, with $\delta : Q \times I \rightarrow Q$, $\lambda : Q \times I \rightarrow O$, and $q \xrightarrow{i/o} q' \Rightarrow \delta(q, i) = q' \wedge \lambda(q, i) = o$.

MAT framework

The most efficient algorithms for model learning all follow the pattern of a *minimally adequate teacher (MAT)* as proposed by Angluin [19]. In the MAT framework, learning is viewed as a game in which a learner has to infer an unknown automaton by asking queries to a teacher. The teacher knows the automaton, which in our setting is a deterministic Mealy machine $\mathcal{M}$. Initially, the learner only knows the inputs I and outputs O of $\mathcal{M}$. The task of the learner is to learn $\mathcal{M}$ through two types of queries:

- With a *membership query*, the learner asks what the response is to an input sequence $\sigma \in I^*$. The teacher answers with the output sequence in $A_{\mathcal{M}}(\sigma)$.
- With an *equivalence query*, the learner asks whether a hypothesized Mealy machine $\mathcal{H}$ is correct, that is, whether $\mathcal{H} \approx \mathcal{M}$. The teacher answers *yes* if this is the case. Otherwise it answers *no* and supplies a *counterexample*, which is a sequence $\sigma \in I^*$ that triggers a different output sequence for both Mealy machines, that is, $A_{\mathcal{H}}(\sigma) \neq A_{\mathcal{M}}(\sigma)$.

Starting from Angluin's seminal L^* algorithm [19], many algorithms have been proposed for learning finite, deterministic Mealy machines via a finite number of queries. We refer to [119] for recent overview. In applications in which one wants to learn a model of a black-box reactive system, the teacher typically consists of a System Under Learning (SUL) that answers the membership queries, and a conformance testing tool [137] that approximates the equivalence queries using a set of *test queries*. A test query consists of asking to the SUL for the response to an input sequence $\sigma \in I^*$, similar to a membership query.

Abstraction

We recall relevant parts of the theory of abstractions from [10]. Existing model learning algorithms are only effective when applied to Mealy machines with small sets of inputs, e.g. fewer than 100 elements. Practical systems like TCP, however, typically have huge alphabets, since inputs and outputs carry parameters of type integer or string. In order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we place a transducer in between the teacher and the learner, which translates concrete inputs in I to abstract inputs in X, concrete outputs in O to abstract outputs in Y, and vice versa. This allows us to abstract a Mealy machine with concrete symbols in I and O to a Mealy machine with abstract symbols in X and Y, reducing the task of the learner to inferring a "small" abstract Mealy machine.

Formally, a *mapper* for inputs I and outputs O is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where I and O are disjoint sets of *concrete input and output symbols*, X and Y are disjoint sets of *abstract input and output symbols*, and $\lambda : R \times (I \cup O) \to (X \cup Y)$, the *abstraction function*, respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \lambda(r, a) \in X$.

Basically, the *abstraction* of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems. Let $\mathcal{M} = \langle I, O, Q, q_0, \to_{\mathcal{M}} \rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta_{\mathcal{A}}, \lambda_{\mathcal{A}} \rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the *abstraction of $\mathcal{M}$ via $\mathcal{A}$*, is the Mealy machine $\langle X, Y \cup \{\perp\}, Q \times R, (q_0, r_0), \to_\alpha \rangle$, where $\perp \notin Y$ is a fresh output and $\to_\alpha$ is given by the rules

$$\frac{q \xrightarrow{i/o}_{\mathcal{M}} q',\ r \xrightarrow{i/x}_{\mathcal{A}} r' \xrightarrow{o/y}_{\mathcal{A}} r''}{(q, r) \xrightarrow{x/y}_{\alpha} (q', r'')} \qquad \frac{\nexists i \in I : r \xrightarrow{i/x}_{\mathcal{A}}}{(q, r) \xrightarrow{x/\perp}_{\alpha} (q, r)}$$

To understand how the mapper is utilized during learning, we follow the execution of a single input of a query. The learner produces an abstract input x, which it sends to the mapper. By inversely following abstraction function $\lambda_{\mathcal{A}}$, the mapper converts this to a concrete input i and updates its state via transition $r \xrightarrow{i/x}_{\mathcal{A}} r'$. The concrete input i is passed on to the teacher, which responds with a concrete output o according to $q \xrightarrow{i/o}_{\mathcal{M}} q'$. This triggers the transition $r' \xrightarrow{o/y}_{\mathcal{A}} r''$ in which the mapper generates the corresponding abstract output y and updates its state again. The abstract output is then returned to the learner.

We notice that the abstraction function is utilized invertedly when translating inputs. More precisely, the abstract input that the learner provides is an output for the mapper.

The translation from abstract to concrete involves picking an arbitrary concrete value that corresponds to the given abstract value. It can be that multiple concrete values can be picked, in which case all values should lead to the same abstract behavior, thus ensuring that the resulting abstract model is deterministic. It can also be that no values correspond to the input abstraction, in which case, by the second rule, $\perp$ is returned to the learner, without consulting the teacher. We define the *abstraction component* implementing $\alpha_{\mathcal{A}}$ as the transducer which follows from the mapper $\mathcal{A}$, but inverts the abstraction of inputs.

From the perspective of a learner, a teacher for $\mathcal{M}$ and abstraction component implementing $\alpha_{\mathcal{A}}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. If $\alpha_{\mathcal{A}}(\mathcal{M})$ is deterministic, then the learner will eventually succeed in learning a deterministic machine $\mathcal{H}$ satisfying $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$. In [10], also a *concretization* operator $\gamma_{\mathcal{A}}$ is defined. This operator is the adjoint of the abstraction operator: it turns any abstract machine $\mathcal{H}$ with symbols in X and Y into a concrete machine with symbols in I and O. If $\mathcal{H}$ is deterministic then $\gamma_{\mathcal{A}}(\mathcal{H})$ is weakly deterministic.

As shown in [10], $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. This tells us that when we apply mapper $\mathcal{A}$ during learning of some "large" Mealy machine $\mathcal{M}$, even though we may not be able to learn the behavior of $\mathcal{M}$ exactly, the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ of the learned abstract model $\mathcal{H}$ is an over-approximation of $\mathcal{M}$, that is, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Similarly to the abstraction component, a *concretization component* for mapper $\mathcal{A}$ implements $\gamma_{\mathcal{A}}$. This component is again fully defined by a mapper, but handles abstraction of outputs invertedly. During model checking, the composition of the abstract model $\mathcal{H}$ and the concretization component for $\mathcal{A}$ provides us with an over-approximation of $\mathcal{M}$.

Framework for mapper definition

In order to apply our abstraction approach, we need an abstraction and a concretization component for a given mapper $\mathcal{A}$. We could implement these components separately in an arbitrary programming language, but then they would have to remain consistent with $\mathcal{A}$. Moreover, ensuring that translation in one component inverts the corresponding translation in the other is non-trivial, and difficult to maintain, as changes in one would have to be applied invertedly in the other.

We used an alternative approach, in which we first define a mapper and then derive the abstraction and concretization components automatically. To this end, we built a language for defining a mapper in terms of (finite) registers, and functions to encode transitions and outputs. Our language supports case distinctions with programming-style if-else-statements, and requires that every branch leads to exactly one output and updates registers exactly once, such that the translations are complete. Except for the restrictions of finiteness and determinism, our language has the expressiveness of a simple programming language and should thus be usable to abstract (and concretize reversely) a wide range of systems and protocols. Listing 2.1 shows the example of a mapper for a simple login system. The mapper stores the first password received, and compares subsequent passwords to it. The abstract passwords used by the learner are {*true*, *false*}, denoting a correct or incorrect password, respectively. At the first attempt, *true* invertedly maps to any concrete password, and *false* maps to ⊥. Later on *true* invertedly maps to the value picked the first time, while *false* maps to any other value. For TCP, we define multiple abstraction functions for inputs and outputs, in terms of multiple parameters per input or output.

Listing 2.1 A simple example mapper for a login system, in a simplified syntax

```
integer $stored := -1$;
map ENTER(integer $password \to$ boolean $correct$)
    if $(stored = -1 \land password \geq 0) \lor stored = password$ then
        $correct := true$
    else
        $correct := false$
update
    if $stored = -1 \land password \geq 0$ then
        $stored := password$
    else
        $stored := stored$                    ▷ Every path explicitly assigns a value
```

To derive the components, we need to implement the inverse of the abstraction function, for both inputs and outputs. This can be achieved using a constraint solver or by randomly picking concrete values until we find one that is translated to the abstract value we want to concretize, where translation is done with the mapper in the forward direction. The latter approach may be hard, as the concrete domain is usually very large, while there may be only a few concrete values matching the abstract value, meaning we would have to test many concrete values before we find a fitting one. To that end, heuristics can help reduce the pool of selectable concrete values. Where the latter approach improves on the former is in testing the abstraction function. We want to ensure that different concrete values translating to the same abstract value lead to the same abstract behavior, as the learner cannot handle non-determinism. A constraint solver usually picks values in a very structured and deterministic way, which does not test the abstraction function well. Picking concrete values randomly and checking the corresponding abstract value allows more control over obtaining a good test coverage, but is in general less scalable.

2.3 Learning Setup

2.3.1 TCP as a System under Learning

In TCP there are two interacting entities, the *server* and the *client*, which communicate over a network through packets, comprising a header and application data. On both sides there is an application, initiating and using the connection through *socket calls*.

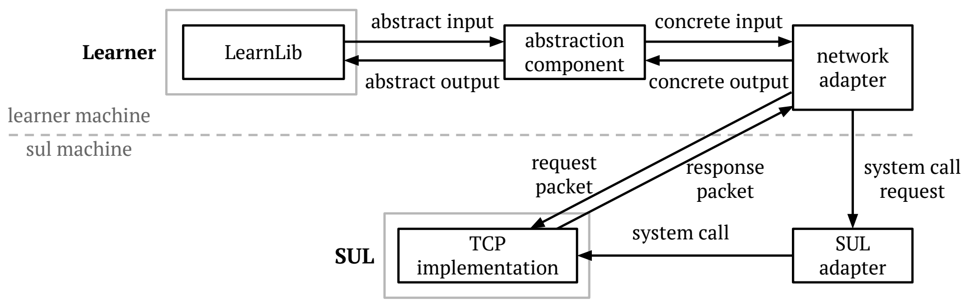


Figure 2.1: Overview of the learning setup.

Each entity is learned separately and is a SUL in the learning context. This SUL thus takes packets or socket calls as inputs. It can output packets or *timeout*, in case the system does not respond with any packet. RFC 793 [176] and its extensions, most notably [44, 168], specify the protocol.

Packets are defined as tuples, comprising sequence and acknowledgement numbers, a payload and flags. By means of abstraction, we reduce the size of sequence and acknowledgement number spaces. Each socket call also defines an abstract and concrete input. Whereas packet configurations are the same for both client and server, socket calls differ. The server can *listen* for connections and *accept* them, whereas the client can actively *connect*. Both parties can *send* and *receive* data, or *close* an established connection (specifically, a half-duplex close [44, p. 88]). The server can additionally *close* its listening socket. Values returned by socket calls are not in the output alphabet to reduce setup complexity.

Figure 2.1 displays the learning setup used. The *learner* generates abstract inputs, representing packets or socket calls. The abstraction component concretizes each input by translating abstract parameters to concrete, and then updates its state. The concrete inputs are then passed on to the *network adapter*, which in turn transforms each input into a packet, sending it directly to the SUL, or into a socket call, which it issues to the SUL *adapter*. The SUL adapter runs on the same environment as the SUL and its sole role is to perform socket calls on the SUL. Each reponse packet generated by the SUL is received by the network adapter, which retrieves the concrete output from the packet or produces a *timeout* output, in case no packet was received within a predefined time interval. The output is then sent to the abstraction component, which computes the abstract output, updates its state again, and sends the abstract output to the learner.

The learner is based on LearnLib [178], a Java library implementing L^* based algorithms for learning Mealy machines. The abstraction component is also written in Java, and interprets and inverts a mapper. The network adapter is a Python program based on Scapy [183], Pcapy [169], and Impacket [118]. It uses Scapy to craft TCP packets, and Scapy together with a Pcapy and Impacket based sniffer to intercept responses. The network adapter is connected to the SUL adapter via a standard TCP connection.

This connection is used for communicating socket calls to be made on the SUL. Finally, the SUL adapter is a program which performs socket calls on command, written in C to have low level access to socket options. The SUL adapter was designed so that it does not block. To that end, it handles blocking calls such as *accept*-call by launching new threads to make these calls. For reasons we explain later, the number of active blocking calls is kept to a maximum of one.

2.3.2 Viewing the SUL as a Mealy Machine

TCP implementations cannot be fully captured by Mealy machines. To learn a model, we therefore need to apply some restrictions. As mentioned, the number of possible values for the sequence and acknowledgement numbers is reduced by means of abstractions. Furthermore, payload is limited to either 0 or 1 byte. Consequently, 1 byte of data is sent upon a *send*-call. Flags are limited to only the most interesting combinations, and we also abstract away from all other fields from the TCP layer or lower layers, allowing Scapy to pick default values.

TCP is also time-dependent. The SUL may, for instance, retransmit packets if they are not acknowledged within a specified time. The SUL may also reset if it does not receive the acknowledgement after a number of such retransmissions, or if it remains in certain states for too long. The former we handled by having the network adapter ignore all retransmissions. For the latter, we verified that the learning queries were short enough so as not to cause these resets.

TCP is inherently concurrent, as a server can simultaneously handle multiple connections. This property is difficult to capture in Mealy machines. To overcome this, the SUL adapter ensures that at most one connection is accepted at any time by using a set of variables for locking and unlocking the *accept* and *connect*-calls. Moreover, at most one active blocking call is allowed at any time, whereas non-blocking socket calls can always be called. The SUL adapter ignores blocking calls when one is already pending, resulting in *timeout* responses.

Furthermore, the backlog size parameter defines the number of connections to be queued up for an eventual *accept*-call by the server SUL. The model grows linearly with this parameter, while only exposing repetitive behavior. For this reason we set the backlog to the value 1.

2.3.3 Technical Challenges

We overcame several technical challenges in order to learn models. Resetting the SUL and setting a proper timeout value are solved similarly to [87].

Our tooling for sniffing packets sometimes missed packets generated by the SUL, reporting erroneous *timeout* outputs. This induced non-deterministic behavior, as a packet may or may not be caught, depending on timing. Each observation is therefore repeated three times to ensure consistency. Consistent outputs are cached to speed up learning, and to check consistency with new observations. It also allows to restart learning with reuse of previous observations.

In order to remove time-dependent behavior we use several TCP settings. Most notably, we disable slow acknowledgements and enable quick acknowledgements where possible (on Linux and FreeBSD). The intuition is that we want the SUL to send acknowledgements whenever they can be issued, instead of delaying them. We also had to disable syn cookies in FreeBSD, as this option caused generation of the initial sequence number in a seemingly time dependent way, instead of using fresh values. For Linux, packets generated by a *send*-call were occasionaly merged with previous unacknowledged packets, so we could only learn a model by omitting *send*-call, although data packets could still be sent from the learner to the SUL.

2.3.4 Mapper Definition

The mapper is based on the work of Aarts et al. [2], and on the RFCs. Socket calls contain no parameters and do not need abstraction, so they are mapped simply with the identity relation. TCP packets are mapped by mapping their parameters individually. Flags are again retained by an identity relation. The sequence and acknowledgement numbers are mapped differently for inputs and outputs; input numbers are mapped to $\{valid, invalid\}$, and outputs are mapped to $\{current, next, zero, fresh\}$. After a connection is set up, the mapper keeps track of the sequence number which should be sent by the SUL and learner. Valid inputs are picked according to this, whereas *current* and *next* represent repeated or incremented numbers, respectively. The abstract output *zero* denotes the concrete number zero, whereas *fresh* is used for all other numbers. If no connection is established, any sequence number is valid (as the RFCs then allow a fresh value), and the only valid acknowledgement number is zero.

Recall that concrete inputs with the same abstract value should lead to an equivalent abstract behavior, otherwise, the behavior exposed to the learner is non-deterministic. As valid behavior is well specified by the RFC's, we were able to define valid inputs based largely on the RFC's.

Invalid inputs, by contrast to their valid counterparts, trigger behaviors that are largely undefined by RFCs. To learn the how the SUL reacts to these inputs, abstractions should be defined precisely according to these behaviors, which is unfeasible to do by hand. As a result, we have excluded invalid inputs from the learning alphabet.

To translate valid inputs, we first used a constraint solver which finds solutions for the transition relation. This is done by taking the disjunction of all path constraints, similar to symbolic execution techniques [130]. However, this did not test the abstraction well, as the constraint solver always picks zero if possible, for example. Hence, we instead randomly picked concrete values until we found one matching the right abstract value. Concrete values were picked with a higher probability if they or their predecessors had been picked or observed previously during the same run. This approach sufficed to translate all values for our experiments.

The concrete mappers developed are available at [2]. They were implemented in the language described in Section 2.2. Note that a mapper is available for every operating system tested, with minor differences between mappers. These differences are the result of adaptations made to fit the the actual implementations. These adaptations concern cases when the SUL is reset by learner inputs. The mapper should be able to detect whenever such a reset occurs and reset its state. Most conditions encoded in the mapper for reset detection are the same among all operating systems. There are however a few conditions specific to certain operating systems. To give an example, receipt of a packet containing the RST flag always indicated reset of the SUL for Linux. By contrast, for FreeBSD and Windows, depending on the flags in the packet which prompted the reset, receipt of a RST might mean that the SUL state remained unaffected. Conditions for detecting SUL reset were refined during experiments. This was possible, since insufficient conditions lead to occurrences of non-determinism, causing learning to fail. We refined the conditions based on these occurrences.

2.3.5 Detailed TCP Mapper Description

Having described the mapper at a high level, we present in detail the actual mapper used for learning Linux TCP stacks. The mappers used for the other operating systems are similar, with the few differences noted earlier.

The mapper features two components, one for processing responses generated by the SUL, the other for processing requests generated by the learner. Responses generated by the SUL are either packets or timeouts. We treat them separately. Requests made by the learner are either socket calls or abstract packet inputs. Socket calls are not processed, as they don't have any parameters nor do they require the mapper to change state. Note that both components as described here perform the transformation from concrete to abstract. However, both can be executed invertedly: during learning the request component is used invertedly for concretizing abstract inputs.

The mapper uses variables to keep track of the sequence numbers of the two interacting sides (during learning, these sides are the SUL and the learner). The variables form the mapper's state and are needed to implement the abstractions. τ is a special value used to initialize/reset variables. It is a small negative value, lying outside of the sequence number domain, hence its use is strictly internal to the mapper. In constructing the mapper, we tried to limit the number of variables, so as to keep its formulation simple. With that being said, our mapper uses the following variables:

- *learnerSeq* - tracks the learner sequence number
- *sulSeq* - tracks the SUL sequence number
- *learnerSeqProposed* - stores the sequence number sent by the learner whenever no sequence number is active at the learner side (*learnerSeq* is τ). We call such a sequence number fresh, as it may be used to establish a new connection. *learnerSeqProposed* is reset upon processing each response.

[2]https://gitlab.science.ru.nl/pfiteraubrostean/tcp-learner/tree/cav-aec/input/mappers

Listing 2.2 Linux mapper: processing incoming packet responses

```
 1: map RESPONSE(flags $flagsIn$, int $concSeq$, int $concAck$, int $concData \to$
                                   flags $absFlagsIn$, absIn $absSeq$, absIn $absAck$)
 2:     $absFlagsIn := flagsIn$
 3:     if $concSeq = sulSeq + 1$ then
 4:         $absSeq := next$
 5:     else
 6:         if $concSeq = sulSeq$ then
 7:             $absSeq := current$
 8:         else
 9:             if $concSeq = 0$ then
10:                 $absSeq := zero$
11:             else
12:                 $absSeq := fresh$
13:     if $concAck = learnerSeq + 1 \vee concAck = learnerSeqProposed + 1$ then
14:         $absAck := next$
15:     else
16:         if $concAck = learnerSeq$ then
17:             $absAck := current$
18:         else
19:             if $concAck = 0$ then
20:                 $absAck := zero$
21:             else
22:                 $absAck := fresh$
23:     update
24:         if $\text{RST} \in flagsIn \vee (learnerSeqProposed \neq \tau \wedge concAck \neq learnerSeqProposed + 1)$ then
25:             $sulSeq := \tau$
26:             $learnerSeq := \tau$
27:         else
28:             if $learnerSeqProposed \neq \tau \vee concSeq = sulSeq + 1$ then
29:                 if $\text{SYN} \in flagsIn \vee \text{FIN} \in flagsIn$ then
30:                     $sulSeq := concSeq$
31:                 else
32:                     if $\text{PSH} \in flagsIn$ then
33:                         $sulSeq := sulSeq + concData$
34:                     else
35:                         $sulSeq := sulSeq$
36:                 $learnerSeq := concAck$
37:             else
38:                 if $\text{SYN} \in flagsIn$ then
39:                     $sulSeq := concSeq$
40:                     if $concAck = zero$ then
41:                         $learnerSeq := learnerSeq$
42:                     else
43:                         $learnerSeq := concAck$
44:                 else
45:                     $sulSeq := sulSeq$
46:                     $learnerSeq := learnerSeq$
47:         $learnerSeqProposed := \tau$
```

Listing 2.2 shows the code used to process packet responses. Note that *concData* stands for the payload size in a response. In lines 2-21 we compare sequence and acknowledgement numbers to mapper variables, resulting in corresponding abstract values. The variables are then updated using these numbers. At a high level, *sulSeq* and *learnerSeq* are preserved unless certain situations occur which result in their change. The code checks specifically for these situations. Lines 23-25 check if connection was reset or could not be set up, in which case *sulSeq* and *learnerSeq* are reset. Lines 27-35 handle the general case when the packet sent by the SUL was not resetting. Therein, *sulSeq* is updated in the presence of countable flags (SYN and FIN) or payload (suggested by PSH). *learnerSeq* always takes the value of the response's acknowledgement number (*concAck*), since through this number, the SUL indicates the sequence number that it next expects from the learner. Lines 37-42 handle the special case when the response contains a SYN flag yet no sequence number was proposed by the learner (this may happen when the SUL actively connects). The response sequence number is stored in *sulSeq* since by including a SYN flag, the SUL announced that it will use this number to establish a connection. The acknowledgment number is only stored in *learnerSeq* if its value is not 0, as 0 indicates that the SUL has no log of the learner's sequence number. In all other cases, *sulSeq* and *learnerSeq* stay the same.

Listing 2.3 Linux mapper: processing outgoing packet requests

```
map REQUEST( flags $flagsOut$, int $concSeq$, int $concAck$, int $concData$ $\rightarrow$
        flags $absFlagsOut$, absOut $absSeq$, absOut $absAck$, int $absData$ )
    $absData := concData,\ absFlagsOut := flagsOut$
    if $learnerSeq = \tau \lor learnerSeq = concSeq$ then
        $absSeq := valid$
    else
        $absSeq := inv$
    if $(sulSeq = \tau \land concAck = 0) \lor (sulSeq \neq \tau \land concAck = sulSeq + 1) \lor \text{ACK} \notin flagsOut$ then
        $absAck := valid$
    else
        $absAck := inv$
    update
        if $\text{RST} \in flagsOut \land absSeq = valid \land absAck = valid$ then
            $learnerSeqProposed := \tau$
            $sulSeq := \tau$
            $learnerSeq := \tau$
        else
            if $learnerSeq = \tau$ then
                $learnerSeqProposed := concSeq$
            else
                $learnerSeqProposed := \tau$
            $sulSeq := sulSeq$
            $learnerSeq := learnerSeq$
```

The code for processing timeout responses (not included) simply resets *learnerSeqProposed* while preserving all other variables. Processing packet requests from the learner (see Listing 2.3) is done in a similar way to responses, only now our abstractions are valid/invalid. A sequence number is valid if either the learner sequence number is not set, in which case any number is valid, or if it is equal to the learner sequence number. All acknowledgement numbers are valid if ACK is not contained in the packet flags as the specification only requires validating acknowledgement numbers if ACK is present. If ACK is contained, then valid numbers are 0 if the SUL sequence number is not set (as there is nothing to acknowledge), or the successor of *sulSeq* otherwise. In the update section, we take care that we update *learnerSeqProposed* accordingly and that we reset the variables whenever valid reset packets are issued. The request component also computes abstraction for the payload size via *absData*. This allows the learner to specify payload size in abstract inputs. We only used payload sizes of 0 or 1.

2.4 Model Learning Results

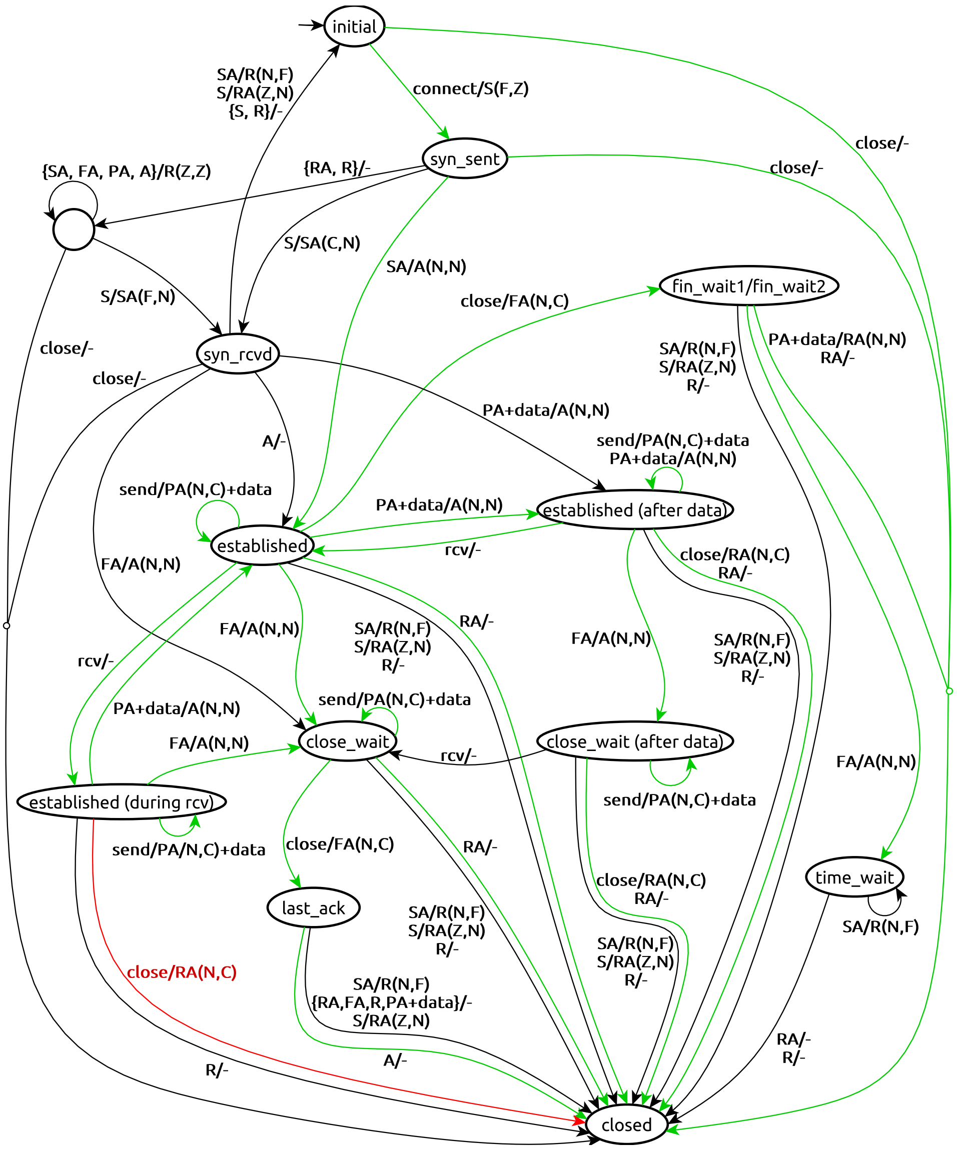


Figure 2.2: Learned model for Windows 8 TCP Client. To reduce the size of the diagram, we eliminate all self loops with *timeout* outputs. We replace flags and abstractions by their capitalized initial letter and hence use S for SYN, A for ACK etc. and N for *next*, C for *current*, Z for *zero* and F for *fresh*. We omit input parameter abstractions, since they are the same for all packets, namely *valid* for both sequence and acknoweldgement numbers. Finally, we group inputs that trigger a transition to the same state with the same output. Timeouts are denoted by '-'.

Using the abstractions defined in Section 2.2, we learned models of the TCP client and server for Windows 8, Ubuntu 14.04 and FreeBSD 10.2. For testing we used the conformance testing algorithm described in [191] to generate efficient test suites which are parameterized by a middle section of length k. Generated exhaustively, these ensure learned model correctness, unless the respective implementation corresponds to a model with at least k more states. For each model, we first executed a random test suite with k of 4, up to 40000 tests for servers, and 20000 tests for clients. We then ran an exhaustive test suite with k of 2 for servers, respectively 3 for clients.

Table 2.1 describes the setting of each of these experiments together with statistics on learning and testing: (1) the number of states in the final model, (2) the number of hypotheses found, (3) the total number of membership queries, (4) the total number of unique test queries run on the SUL before the last hypothesis, (5) the number of unique test queries run to validate the last hypothesis. The models learned and other experimental data such as input configurations used, hypothesis models generated during learning and statistics, are available at [3].

SUL		**States**	**Hyp.**	**Mem. Q.**	**Tests to l. Hyp.**	**Tests on l. Hyp.**
Client	Windows 8	13	2	1576	1322	50243
Server	Windows 8	38	10	11428	9549	65040
Client	Ubuntu 14.04	15	2	1974	15268	56174
Server	Ubuntu 14.04	57	14	17879	15681	66523
Client	FreeBSD 10.2	12	2	1456	1964	47387
Server	FreeBSD 10.2	55	18	22287	12084	75894

Table 2.1: Statistics for learning experiments

Figure 2.2 shows the model learned for the Windows 8 client. This model covers standard client behavior, namely connection setup, sending and receiving data and connection termination. Based on predefined access sequences, we identify each state with its analogous state in the RFC state diagram [176, p. 23], if such a state exists. Transitions taken during simulated communication between a Windows client and a server are colored green. These transitions were identified during model checking, on which we expand in Section 2.5. Models for Linux and FreeBSD clients can be found in the appendix at the end of this chapter.

Table 2.1 shows that the models for the Linux and FreeBSD servers have more states than for Windows, and all models have more states than described in the specification. We attribute this to several factors. We have already mentioned that model sizes grow linearly with the value of the backlog-parameter. While we set it to 1, the setting is overridden by operating system imposed minimum value of 2 for FreeBSD and Linux. Moreover, SUL behavior depends on blocking system calls and on whether the receive buffer is empty or not. Although specified, this is not modeled explicitly in the specification state diagram. As an example, the ESTABLISHED and CLOSE WAIT states from the standard each have multiple corresponding states in the model in Figure 2.2.

[3]https://gitlab.science.ru.nl/pfiteraubrostean/tcp-learner/tree/cav-aec/models

Non-conformance of implementations

Inspection of learned models revealed several cases of non-conformance to RFC's in the corresponding implementations.

A first non-conformance involves terminating an established connection with a CLOSE. The resulting output should contain a FIN if the closing side has read via RCV all data received from the other side. Note that data is first received in a side's buffer, from which it is read and removed by RCV calls. If there is received data not yet read by the closing side, the output should contain a RST, which would signal to the other side an aborted termination [44, p. 88]. Windows does not conform to this, as a CLOSE can generate a RST instead of a FIN even in cases where there is no data to be read (the data buffer is empty), namely, in states where a RCV call is pending. Figure 2.2 marks this behavior in red. FreeBSD implementations are also non-compliant, as they always generate FIN packets on a CLOSE, regardless if all data has been read. This would arguably fall under the list of common bugs [168], namely "Failure to send a RST after Half Duplex Close". The learned Linux models fully comply to these specifications.

A second non-conformance has to do with the processing of SYN packets. On receiving a SYN packet in a synchronized state, if the sequence number is in "the window" (as it always is, in our case), the connection should be reset (via a corresponding RST packet) [176, p.71]. Linux implementations conform for SYN packets but not for SYN+ACK packets, to which they respond by generating an acknowledgement with no change of state. Both Windows and FreeBSD respect this specification.

We note a final non-conformance in Windows implementations. In case the connection does not exist (CLOSED), a reset should be sent in response to any incoming packet except for another reset [176, p. 36], but Windows 8 sends nothing. FreeBSD can be configured to respond in a similar way to Windows, by changing the *blackhole setting*.[4] This behavior is claimed to provide "some degree of protection against stealth scans", and is thus intentional.

2.5 Model Checking Results

2.5.1 Model Checking the Learned Behavior

We analyzed the learned models of TCP implementations using the model checker NuSMV [66]. We composed pairs of learned client and server models with a hand-made model of a non-lossy network, which simply delivers output from one entity as input for the other entity. Since the abstract input and output domains are different, the abstract models cannot communicate directly, and so we had to encode the concretized models within NuSMV code. We wrote a script that translated the abstract Mealy machine models from LearnLib to NuSMV modules, and another script that translated

[4]https://www.freebsd.org/cgi/man.cgi?query=blackhole

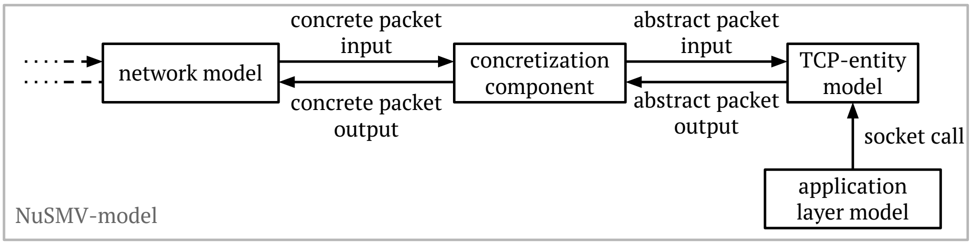


Figure 2.3: Schematic overview of NuSMV-model. Only half of the setup is shown in detail, as the model is symmetric and another TCP-entity model is connected to the network.

the corresponding mappers to NuSMV modules. TCP entities produce abstract outputs, which are translated to concrete. The network module then passes along such concrete messages. Before being delivered to the other entity, these messages are again transformed into abstract inputs. By encoding mapper functions as relations, NuSMV is able to compute both the abstraction function and its inverse, i.e., act as a concretization component. The global structure of the model is displayed in Figure 2.3.

In Mealy machines, transitions are labeled by an input/output pair. In NuSMV transitions carry no labels, and we also had to split the Mealy machine transitions into a separate input and output part in order to enable synchronization with the network. Thus, a single transition $q \xrightarrow{i/o} q'$ from a (concrete) Mealy machine is translated to a pair of transitions in NuSMV:

$$(loc = q, in = .., out = ..) \rightarrow (loc = q, in = i, out = ..) \rightarrow (loc = q', in = i, out = o).$$

Sequence and acknowledgement numbers in the implementations are 32-bit numbers, but were restricted to 3-bit numbers to reduce the state space. Whereas concrete messages are exchanged from one entity to the other, socket call inputs from the application are simulated by allowing system-calls to occur non-deterministically. A simplification we make is that we do not allow parallel actions: an action and all resulting packets have to be fully processed until another action can be generated. Consequently, there can be at most one packet in the composite model at any time. For example, once a three way handshake is initiated between a client and a listening server via a *connect*-call, no more system-calls can be performed until the handshake is finalized.

2.5.2 Checking Specifications

After a model is composed, the interaction between TCP entities can be analyzed using the NuSMV model checker. However, it is important to realize that, since we used abstractions, the learned models of TCP servers and clients are over-approximations of the actual behaviors of these components. If Mealy machine $\mathcal{M}$ models the actual behavior of a component, $\mathcal{A}$ is the mapper used, and $\mathcal{H}$ is the abstract model that we learned then, as explained in Section 2.2, correctness of $\mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Since $\gamma_{\mathcal{A}}(\mathcal{H})$ is weakly deterministic, in this case there exists a forward simulation relation from $\mathcal{M}$ to $\gamma_{\mathcal{A}}(\mathcal{H})$. This forward simulation is preserved by the translation from Mealy machines to NuSMV. Results from Grumberg and Long [100] then imply that, for any $\forall$CTL*-formula (which includes all LTL-formulas) we can transfer model checking results for $\gamma_{\mathcal{A}}(\mathcal{H})$ to the (unknown) model $\mathcal{M}$. Since simulations are preserved by composition, this result even holds when $\gamma_{\mathcal{A}}(\mathcal{H})$ is used as a component in a larger model.

Another essential point is that only a subset of the abstract inputs is used for learning. Hence invalid inputs (i.e. inputs with invalid parameters) are not included in our models. Traces with these inputs can therefore not be checked. Hence, the first property that we must check is a global invariant that asserts that invalid inputs will never occur. In case they do, NuSMV will provide a counterexample, which is used to find the cause of invalidity. During our initial experiments, NuSMV found several counterexamples showing that invalid inputs may occur. Based on analysis of these counterexamples we either refined/corrected the definition of one of the mappers, or we discovered a counterexample for the correctness of one of the abstract models. After a number of these iterations, we obtained a model in which invalid inputs can no longer occur. As mapper construction is done manually, these iterations are also not yet automated.

With only valid inputs, the composite model may be checked for arbitrary $\forall$CTL* formulas. Within these formulas, we may refer to input and output packets and their constituents (sequence numbers, acknowledgements, flags,..). This yields a powerful language for stating properties, illustrated by a few examples below. These formulas are directly based on the RFC's.

Many properties that are stated informally in the RFC's refer to control states of the protocol. These control states, however, cannot be directly observed in our black-box setting. Nevertheless, we can identify states, e.g. based on inputs and outputs leading to and from it. For example, we base the proposition *established* on RFC 793, which states that: "The connection becomes 'established' when sequence numbers have been synchronized in both directions" [176, p. 11], and that only a CLOSE or ABORT socket call or incoming packets with a RST or FIN can make an entity leave the ESTABLISHED state [176, section 3.9].

We first show a simple safety formula checking desynchonization: if one entity is in the ESTABLISHED state, the other cannot be in SYN_SENT and TIME_WAIT:

$$\mathsf{G}\neg(\textit{tcp1-state} = \textit{established} \wedge (\textit{tcp2-state} = \textit{syn\_sent} \vee \textit{tcp2-state} = \textit{time\_wait}))$$

The next specification considers terminating an established connection with a CLOSE-input. The output should contain a FIN, except if there is unread data (in which case it should contain a RST). This corresponds to the first non-conformance case explained in Section 2.4. The specification is captured by the following formula, in which T is the triggered-operator as defined in NuSMV.

$$\mathsf{G}(\mathit{state} = \mathit{established} \rightarrow ((\mathit{input} = \mathit{rcv}\ \mathsf{T}\ \mathit{input} \neq \mathit{packet\ with\ data}) \wedge \mathit{input} = \mathit{close}) \rightarrow (\mathsf{F}\ \mathit{output} = \mathit{packet\ with}\ \textsc{fin})))$$

We have formalized and checked, in a similar way, specifications for all other non-conforming cases as well as many other specifications.

We have also checked which transitions in the abstract models are reachable in the composed system. For every transition, we take its input and starting state, and check whether they can occur together. In this way we can find the reachable parts of model. This proves useful when analyzing models, as the reachable parts likely harbor bugs with the most impact. Similarly, comparing reachable parts helps reveal the most relevant differences between implementations. The first and third non-conformances in Section 2.4 occur in the reachable parts of the respective models. Figure 2.2 marks these parts in green.

2.6 Conclusions and Future Work

We combined model learning, model checking and abstraction techniques to obtain and analyze models of Windows, Linux and FreeBSD TCP server and client implementations. Composing these models together with the model of a network allowed us to perform model checking over the composite setup and verify that any valid number generated by one TCP entity is seen as valid number by the other TCP entity. We have also identified breaches of the RFC's in all operating systems, and confirmed them by formulating temporal specification and checking them. Work in this chapter suggests several directions for future work.

Based on our understanding of TCP, we manually defined abstractions (mappers) that made it possible to learn models of TCP implementations. Getting the mapper definitions right turned out to be tricky. In fact, we had to restrict our learning experiments to *valid* abstractions of the sequence and acknowledgement numbers. This proved limiting when searching for interesting rules to model check, like for example those that would expose known implementation bugs. Such rules often concern invalid parameters, which do not appear in the models we learned. Additionaly, we had to manually refine our mapper due to counterexamples found by the model checker. Learning algorithms that construct the abstractions automatically could potentially solve this problem. We hope that extensions of the learning algorithms for register automata as implemented in the Tomte [5] and RALib [53] tools will be able to construct abstractions for TCP fully automatically.

Work in this chapter was severely restricted by the lack of expressivity of Mealy machines. In order to squeeze the TCP implementation into a Mealy machine, we had to eliminate timing based behavior as well as re-transmissions. Other frameworks for modeling state machines might facilitate modeling these aspects. Obviously, we would also need learning algorithms capable of generating such state machines.

There has been some preliminary work on extending learning algorithms to timed automata [98, 209], and to I/O transition systems [12, 210], with the additional benifit of approximate learning. Approximate learning learns an upper and lower boundary to the behavior of the system, instead of an exact model. This may allow to abstract away the corner-cases in the model and the mapper, if they are not relevant for the specifications. Extensions of this work could eliminate some of the restrictions that we encountered.

2.A Model of Linux TCP Client

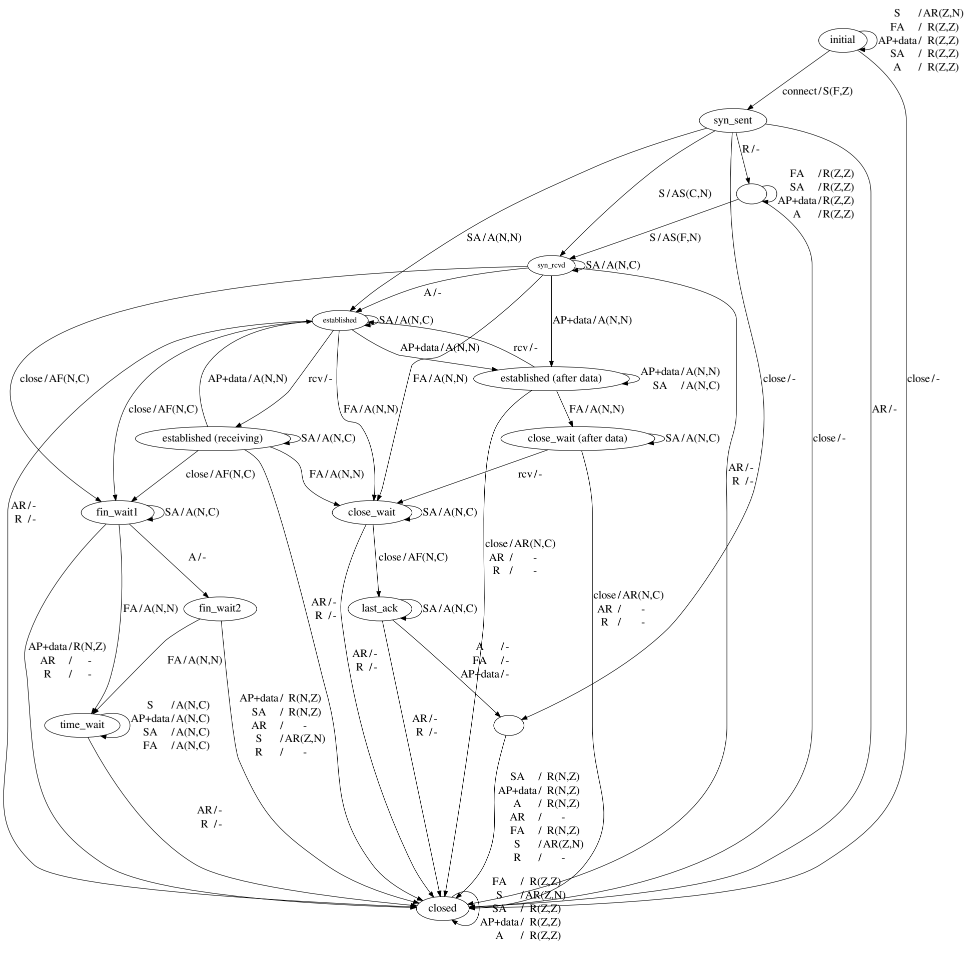


Figure 2.4: Learned model for Linux TCP Client. We use all the pruning strategies that we used for Window 8 apart from merging transitions with the same output.

2.B Model of FreeBSD TCP Client

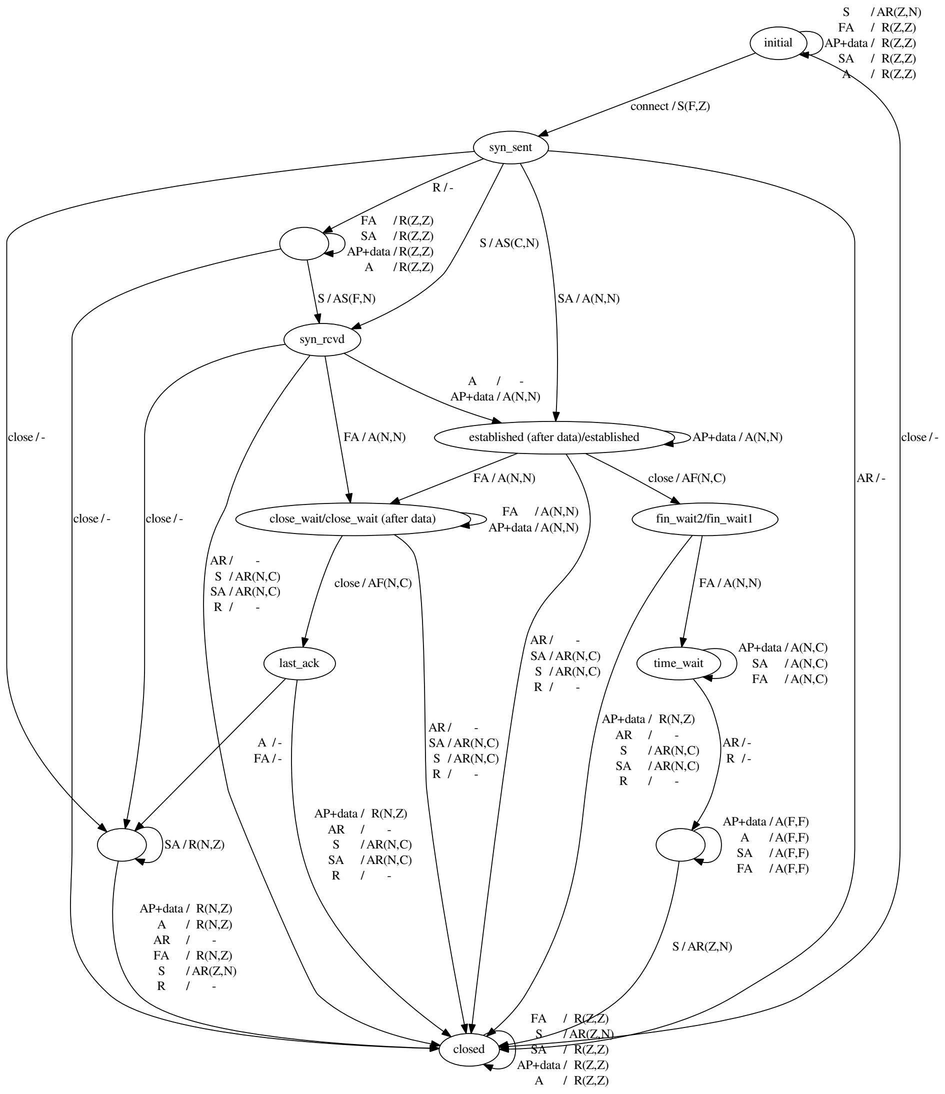


Figure 2.5: Learned model for FreeBSD TCP Client. We use all the pruning strategies that we used for Window 8 apart from merging transitions with the same output.

Chapter 3

Model Learning and Model Checking of SSH Implementations

We apply model learning on three SSH implementations to infer state machine models, and then use model checking to verify that these models satisfy basic security properties and conform to the RFCs. Our analysis showed that all tested SSH server models satisfy the stated security properties, but uncovered several violations of the standard.

3.1 Introduction

SSH is a security protocol that is widely used to interact securely with remote machines. The Transport layer of SSH has been subjected to security analysis [219], incl. analyses that revealed cryptographic shortcomings [18, 25, 167].

Whereas these analyses consider the abstract cryptographic protocol, this chapter looks at actual implementations of SSH, and investigates flaws in the program logic of these implementations, rather than cryptographic flaws. Such logical flaws have occurred in implementations of other security protocols, notably TLS, with Apple's 'goto fail' bug and the FREAK attack [32]. For this we use model learning (a.k.a. active automata learning) [19, 170, 205] to infer state machines of three SSH implementations, which we then analyze by model checking for conformance to both functional and security properties.

The properties we verify for the inferred state machines are based on the RFCs that specify SSH [221–224]. These properties are formalized in LTL and verified using NuSMV [66]. We use a model checker since the models are too complex for manual inspection (they are trivial for NuSMV). Moreover, by formalizing the properties we can better assess and overcome vagueness or under-specification in the RFC standards.

This chapter is born out of two recent theses [138, 208], and is to our knowledge the first combined application of model learning and model checking in verifying

SSH implementations, or more generally, implementations of any network security protocol.

Related work Chen et al. [60] use the MOPS software model checking tool to detect security vulnerabilities in the OpenSSH C implementation due to violation of folk rules for the construction of secure programs such as "Do not open a file in writing mode to stdout or stderr". Udrea et al. [203] also investigated SSH implementations for logical flaws. They used a static analysis tool to check two C implementations of SSH against an extensive set of rules. These rules not only express properties of the SSH protocol logic, but also of message formats and support for earlier versions and various options. Our analysis only considers the protocol logic. However, their rules were tied to routines in the code, so had to be slightly adapted to fit the different implementations. In contrast, our properties are defined at an abstract level so do not need such tailoring. Moreover, our black-box approach means we can analyze any implementation of SSH, not just open-source C implementations.

Formal models of SSH in the form of state machines have been used before, namely for a manual code review of OpenSSH [174], formal program verification of a Java implementation of SSH [173], and for model based testing of SSH implementations [39]. All this research only considered the SSH Transport layer, and not the other SSH protocol layers.

Model learning has previously been used to infer state machines of EMV bank cards [3], electronic passports [11], hand-held readers for online banking [59], and implementations of TCP [88] and TLS [181]. Some of these studies relied on manual analysis of learned models [3, 11, 181], but some also used model checkers [59, 88].

Instead of using active learning as we do, it is also possible to use passive learning to obtain protocol state machines [218]. Here network traffic is observed, and not actively generated. This can then provide a probabilistic characterization of normal network traffic, but it cannot uncover implementation flaws that occur in strange message flows, which is our goal.

3.2 Model Learning

3.2.1 Mealy Machines

A *Mealy machine* is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where I is a finite set of inputs, O is a finite set of outputs, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times I \to Q$ is a transition function, and $\lambda : Q \times I \to O$ is an output function. Output function λ is extended to sequences of inputs by defining, for all $q \in Q$, $i \in I$ and $\sigma \in I^*$, $\lambda(q, \epsilon) = \epsilon$ and $\lambda(q, i\sigma) = \lambda(q, i)\lambda(\delta(q, i), \sigma)$. The behavior of Mealy machine $\mathcal{M}$ is defined by function $A_{\mathcal{M}} : I^* \to O^*$ with $A_{\mathcal{M}}(\sigma) = \lambda(q^0, \sigma)$, for $\sigma \in I^*$. Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are *equivalent*, denoted $\mathcal{M}_1 \approx \mathcal{M}_2$, iff $A_{\mathcal{M}_1} = A_{\mathcal{M}_2}$. Sequence $\sigma \in I^*$ *distinguishes* $\mathcal{M}_1$ and $\mathcal{M}_2$ if and only if $A_{\mathcal{M}_1}(\sigma) \neq A_{\mathcal{M}_2}(\sigma)$.

3.2.2 MAT Framework

The most efficient algorithms for model learning (see [119] for a recent overview) all follow the pattern of a *minimally adequate teacher (MAT)* as proposed by Angluin [19]. Here learning is viewed as a game in which a *learner* has to infer an unknown automaton by asking queries to a teacher. The teacher knows the automaton, which in our setting is a Mealy machine $\mathcal{M}$, also called the System Under Learning (SUL). Initially, the learner only knows the input alphabet I and output alphabet O of $\mathcal{M}$. The task of the learner is to learn $\mathcal{M}$ via two types of queries:

- With a *membership query*, the learner asks what the response is to an input sequence $\sigma \in I^*$. The teacher answers with the output sequence in $A_{\mathcal{M}}(\sigma)$.
- With an *equivalence query*, the learner asks whether a hypothesized Mealy machine $\mathcal{H}$ is correct, that is, whether $\mathcal{H} \approx \mathcal{M}$. The teacher answers *yes* if this is the case. Otherwise it answers *no* and supplies a *counterexample*, which is a sequence $\sigma \in I^*$ that triggers a different output sequence for both Mealy machines, that is, $A_{\mathcal{H}}(\sigma) \neq A_{\mathcal{M}}(\sigma)$.

The MAT framework can be used to learn black-box models of software. If the behavior of a software system, or System Under Learning (SUL), can be described by some unknown Mealy machine $\mathcal{M}$, then a membership query can be implemented by sending inputs to the SUL and observing resulting outputs. An equivalence query can be approximated using a conformance testing tool [137] via a finite number of *test queries*. A test query consists of asking the SUL for the response to an input sequence $\sigma \in I^*$, similar to a membership query. Note that this cannot rule out that there is more behavior that has not been discovered.

3.2.3 Abstraction

Most current learning algorithms are only applicable to Mealy machines with small alphabets comprising abstract messages. Practical systems typically have parameterized input/output alphabets, whose application triggers updates on the system's state variables. To learn these systems we place a *mapper* between the learner and the SUL. The MAPPER is a transducer which translates concrete inputs to abstract inputs and concrete outputs to abstract outputs. For a thorough discussion of mappers, we refer to [10].

3.3 The Secure Shell Protocol

The Secure Shell Protocol (or SSH) is a protocol used for secure remote login and other secure network services over an insecure network. It runs as an application layer protocol on top of TCP, which provides reliable data transfer, but does not provide any form of connection security. The initial version of SSH was superseded by a second

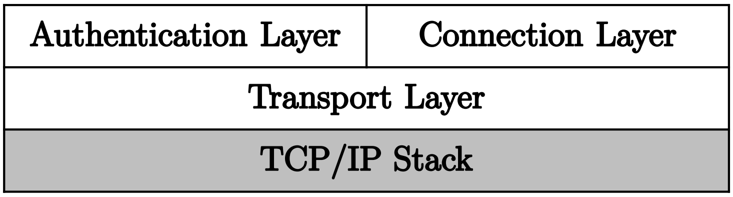


Figure 3.1: SSH protocol layers

version (SSHv2), after the former was found to contain design flaws which could not be fixed without losing backwards compatibility [93]. This work focuses on SSHv2.

SSHv2 follows a client-server paradigm. The protocol consists of three layers (Figure 3.1):

1. The *transport layer protocol* (RFC 4253 [224]) forms the basis for any communication between a client and a server. It provides confidentiality, integrity and server authentication as well as optional compression.
2. The *authentication protocol* (RFC 4252 [221]) is used to authenticate the client to the server.
3. The *connection protocol* (RFC 4254 [222]) allows the encrypted channel to be multiplexed in different channels. These channels enable a user to run multiple applications, such as terminal emulation or file transfer, over a single SSH connection.

Each layer has its own specific messages. The SSH protocol is interesting in that outer layers do not encapsulate inner layers, and different layers can interact. For this reason, we opt to analyze SSH as a whole, instead of analyzing its constituent layers independently. Below we discuss each layer, outlining the relevant messages which are later used in learning, and characterizing the so-called *happy flow* that a normal protocol run follows.

At a high level, a typical SSH protocol run uses the three constituent protocols in the order given above: after the client establishes a TCP connection with the server, (1) the two sides use the Transport layer protocol to negotiate key exchange and encryption algorithms, and use these to establish session keys, which are then used to secure further communication; (2) the client uses the user authentication protocol to authenticate to the server; (3) the client uses the connection protocol to access services on the server, for example the terminal service.

3.3.1 Transport Layer

SSH runs over TCP, and provides end-to-end confidentiality and integrity using session keys. Once a TCP connection has been established with the server, these session keys are securely negotiated using a *key exchange* algorithm, the first step of the protocol. The key exchange begins by the two sides exchanging their preferences for the key exchange algorithm to be used, as well as encryption, compression and hashing algorithms. Preferences are sent with a KEXINIT message. Subsequently, key exchange using the negotiated algorithm takes place. Following this algorithm, one-time session keys for encryption and hashing are generated by each side, together with an identifier for the session. The main key exchange algorithm is Diffie-Hellman, which is also the only one required by the RFC. For the Diffie-Hellman scheme, KEX30 and KEX31 are exchanged to establish fresh session keys. These keys are used from the moment the NEWKEYS command has been issued by both parties. A subsequent SR_AUTH requests the authentication service. The happy flow thus consists of the succession of the three steps comprising key exchange, followed up by a successful authentication service request. The sequence is shown in Figure 3.2.

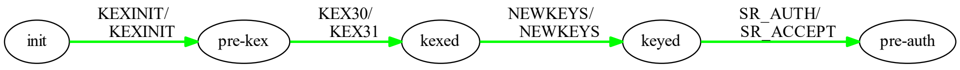


Figure 3.2: The happy flow for the Transport layer.

Key re-exchange [224, p. 23], or *rekeying*, is an almost identical process, the difference being that instead of taking place at the beginning, it takes place once session keys are already in place. The purpose is to renew session keys so as to foil potential replay attacks [223, p. 17]. It follows the same steps as key exchange. A fundamental property of rekeying is that it should preserve the state; that is, after the rekeying procedure is completed, the protocol should be in the same state as it was before the rekeying started. The only thing that changes is that the new keys negotiated through rekeying are now used, instead of the old ones.

3.3.2 Authentication Layer

Once a secure tunnel has been established, the client can authenticate. For this, four authentication methods are defined in RFC 4252 [221]: password, public-key, host-based and none. The authentication request includes a user name, service name and authentication data, which consists of both the authentication method as well as the data needed to perform the actual authentication, such as the password or public key. The happy flow for this layer, as shown in Figure 3.3, is simply a single protocol step that results in a successful authentication. The messages UA_PW_OK and UA_PK_OK achieve this for respectively password and public key authentication.

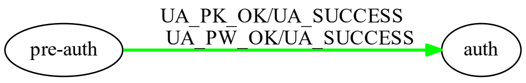


Figure 3.3: The happy flow for the user Authentication layer.

3.3.3 Connection Layer

Successful authentication makes services of the Connection layer available. The Connection layer enables the user to open and close channels of various types, with each type providing access to specific services. Of the various services available, we focus on the remote terminal over a session channel, a classical use of SSH. The happy flow consists of opening a session channel, CH_OPEN, requesting a "pseudo terminal" CH_REQUEST_PTY, optionally sending and managing data via the messages CH_SEND_DATA, CH_WINDOW_ADJUST, CH_SEND_EOF, and eventually closing the channel via CH_CLOSE, as depicted in Figure 3.4.

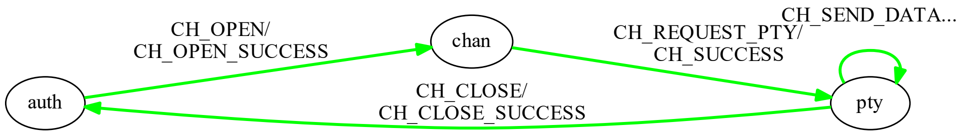


Figure 3.4: The happy flow for the Connection layer.

3.4 The Learning Setup

The learning setup consists of three components: the learner, the MAPPER and the SUL. The learner generates abstract inputs, representing SSH messages. The MAPPER transforms these messages into well-formed SSH packets and sends them to the SUL. The SUL sends response packets back to the MAPPER, which in turn, translates these packets to abstract outputs. The MAPPER then sends the abstract outputs back to the learner.

The learner uses LearnLib [178], a Java library implementing L^* based algorithms for learning Mealy machines. The MAPPER is based on Paramiko, an open-source SSH implementation written in Python[1]. We opted for Paramiko because its code is relatively well structured and documented. The SUL can be any existing implementation of an SSH server. The three components communicate over sockets, as shown in Figure 3.5.

SSH is a complex client-server protocol. Work in this chapter is therefore concentrated on learning models of the implementation of the server, and not of the client. We

[1]Paramiko is available at http://www.paramiko.org/

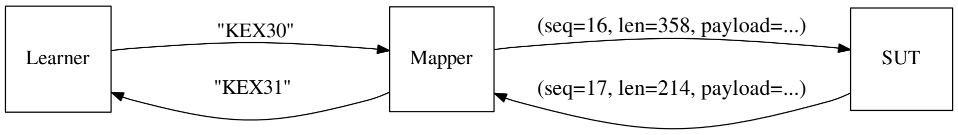

Learner
"KEX30"
"KEX31"
Mapper
(seq=16, len=358, payload=...)
(seq=17, len=214, payload=...)
SUT

Figure 3.5: The SSH learning setup.

further restrict learning to only exploring the terminal service of the Connection layer, as we consider it to be the most interesting from a security perspective. Algorithms for encryption, compression and hashing are left to default settings and are not purposefully explored. Also, the starting state of the SUL is one where a TCP connection has already been established and where SSH versions have been exchanged, which are prerequisites for starting the Transport layer protocol.

3.4.1 The Learning Alphabet

The alphabet we use consists of inputs, which correspond to messages sent to the server, and outputs, which correspond to messages received from the server. We split *the input alphabet* into three parts, one for each of the protocol layers.

Learning does not scale with a growing input alphabet, and since we are only learning models of servers, we remove those inputs that are not intended to ever be sent to the server[2]. Furthermore, from the Connection layer we only use messages for channel management and the terminal functionality. Finally, because we will only explore protocol behavior after SSH versions have been exchanged, we exclude the messages for exchanging version numbers.

The resulting lists of inputs for the three protocol layers are given in Tables 3.1-3.3. In some experiments, we used only a subset of the most essential inputs, to further speed up experiments. This *restricted alphabet* significantly decreases the number of queries needed for learning models while only marginally limiting explored behavior. We discuss this again in Section 3.5. Inputs included in the restricted alphabet are marked by '*' in the tables below.

Table 3.1 lists the Transport layer inputs. We include a version of the KEXINIT message with `first_kex_packet_follows` disabled. This means no guess [224, p. 17] is attempted on the SUL's parameter preferences. Consequently, the SUL will have to send its own KEXINIT in order to convey its own parameter preferences before key exchange can proceed. Also included are inputs for establishing new keys (KEX30, NEWKEYS), disconnecting (DISCONNECT), as well as the special inputs IGNORE, UNIMPL and DEBUG. The latter are not interesting, as they are normally ignored by implementations. Hence they are excluded from our restricted alphabet. DISCONNECT could take a long time to execute, so was also excluded.

The Authentication layer defines a single client message type for the authentication requests [221, p. 4]. Its parameters contain all information needed for authentication. Four authentication methods exist: none, password, public key and host-based. Our mapper supports all methods except host-based authentication because some SUTs don't support this feature. Both the public key and password methods have OK and NOK variants, which provide respectively correct and incorrect credentials. Our

[2]This means we exclude the messages SERVICE_ACCEPT, UA_ACCEPT, UA_FAILURE, UA_BANNER, UA_PK_OK, UA_PW_CHANGEREQ, CH_SUCCESS and CH_FAILURE from our alphabet.

Table 3.1: Transport layer inputs

Message	**Description**
DISCONNECT	Terminates the current connection [224, p. 23]
IGNORE	Has no intended effect [224, p. 24]
UNIMPL	Intended response to unrecognized messages [224, p. 25]
DEBUG	Provides other party with debug information [224, p. 25]
KEXINIT*	Sends parameter preferences [224, p. 17]
KEX30*	Initializes the Diffie-Hellman key exchange [224, p. 21]
NEWKEYS*	Requests to take new keys into use [224, p. 21]
SR_AUTH*	Requests the authentication protocol [224, p. 23]
SR_CONN*	Requests the connection protocol [224, p. 23]

restricted alphabet supports only public key authentication, as the implementations processed this faster than the other authentication methods.

Table 3.2: Authentication layer inputs

Message	**Description**
UA_NONE	Authenticates with the "none" method [221, p. 7]
UA_PK_OK*	Provides a valid name/key pair [221, p. 8]
UA_PK_NOK*	Provides an invalid name/key pair [221, p. 8]
UA_PW_OK	Provides a valid name/password pair [221, p. 10]
UA_PW_NOK	Provides an invalid name/password pair [221, p. 10]

The Connection layer allows clients to manage channels and request services over them. In accordance with our learning goal, our mapper only supports inputs for requesting terminal emulation, plus inputs for channel management as shown in Table 3.3. The restricted alphabet only supports the most general channel management inputs, and excludes those not expected to produce state change.

Table 3.3: Connection layer inputs

Message	**Description**
CH_OPEN*	Opens a new channel [222, p. 5]
CH_CLOSE*	Closes a channel [222, p. 9]
CH_EOF*	Indicates that no more data will be sent [222, p. 9]
CH_DATA*	Sends data over the channel [222, p. 7]
CH_EDATA	Sends typed data over the channel [222, p. 8]
CH_WINDOW_ADJUST	Adjusts the window size [222, p. 7]
CH_REQUEST_PTY*	Requests terminal emulation [222, p. 11]

The output alphabet includes all messages an SSH server generates, which may include, with identical meaning, any of the messages defined as inputs. This also includes responses to various requests: KEX31 [224, p. 21] as reply to KEX30, SR_SUCCES in response to service requests (SR_AUTH and SR_CONN), UA_SUCCESS and UA_FAILURE [221, p.

5,6] in response to authentication requests, and
CH_OPEN_SUCCESS [222, p. 6] and CH_SUCCESS [222, p. 10] , in positive response to CH_OPEN and CH_REQUEST_PTY respectively. To these outputs, we add NO_RESP for when the SUL generates no output, and the special outputs CH_NONE, CH_MAX and NO_CONN, and BUFFERED, which we discuss in the next subsections.

3.4.2 The Mapper

The MAPPER must provide a translation between abstract messages and well-formed SSH messages: it has to translate abstract inputs listed in Tables 3.1-3.3 to actual SSH packets, and translate the SSH packets received in answer to our abstract outputs. If no answer is received on an input, the MAPPER must return an output indicating timeout, which in our case is the NO_RESP message.

The sheer complexity of the MAPPER meant that it was easier to adapt an existing SSH implementation, rather than construct the MAPPER from scratch. After all, in many ways the MAPPER acts similar to an SSH client. Paramiko already provides mechanisms for encryption/decryption, as well as routines for constructing and sending the different types of packets, and for receiving them. These routines are called by control logic dictated by Paramiko's own state machine. The MAPPER was constructed by replacing this control logic with one dictated by messages received from the learner. The technical nature of the MAPPER and the fact that it was adapted from an existing codebase makes it difficult to formalize accurately. We hence only give an informal description of its behavior.

The MAPPER maintains a set of state variables to record parameters of the ongoing session, including the server's preferences for key exchange and encryption algorithm, parameters of these protocols, and, once it has been established, the session key. These parameters are updated when receiving messages from the server and used to concretize inputs to actual SSH messages to the server.

For example, upon receiving a KEXINIT from the SUL, the MAPPER saves the SUL's preferences for key exchange, hashing and encryption algorithms. Initially these parameters are all set to the defaults that any server should support, as required by the RFC. On receiving KEX31 in response to the KEX30 input, the MAPPER saves the hash, as well as the new keys. Finally, a NEWKEYS response prompts the MAPPER to use the new keys negotiated earlier in place of the older ones, if such existed.

The MAPPER also contains a buffer for storing opened channels, which is initially empty. On a CH_OPEN from the learner, the MAPPER adds a channel to the buffer with a randomly generated channel identifier; on a CH_CLOSE, it removes the channel (if there was any). The buffer size, or the maximum number of opened channels, is limited to one. Initially the buffer is empty. The MAPPER also stores the sequence number of the last received message from the SUL. This number is then used when constructing UNIMPL inputs.

In the following cases, inputs are answered by the MAPPER directly instead of being sent to the SUL to find out its response: (1) on receiving a CH_OPEN input if the buffer has reached the size limit, the MAPPER directly responds with CH_MAX; (2) on receiving any input operating on a channel (all Connection layer inputs other than CH_OPEN) when the buffer is empty, the MAPPER directly responds with CH_NONE; (3) if connection with the SUL was terminated, the MAPPER responds with a NO_CONN message, as sending further messages to the SUL is pointless in that case.

3.4.3 Practical Complications

SSH implementations even behind the MAPPER abstraction may not behave like deterministic Mealy machines, a prerequisite for the learning algorithm to succeed. Sources of non-determinism are:

1. Underspecification in the SSH specification (for example, by not specifying the order of certain messages) allows some non-deterministic behavior. Even if client and server do implement a fixed order for messages they sent, the asynchronous nature of communication means that the interleaving of sent and received messages may vary. Moreover, client and server are free to intersperse DEBUG and IGNORE messages at any given time[3]
2. Timing is another source of non-deterministic behavior. For example, the MAPPER might time-out before the SUL had sent its response. Some SULs also behave unexpectedly when a new input is received too shortly after the previous one. Hence in our experiments we adjusted time-out periods accordingly so that neither of these events occur, and the SUL behaves deterministically all the time.

To detect non-determinism, the MAPPER caches all observations in an SQLite database and verifies if new observations are consistent with previous ones. If not, it raises a warning, which then needs to be manually investigated. We analyzed each warning until we found a setting under which behavior was deterministic.

The cache also acts as a cheap source of responses for already answered queries. Finally, by re-loading the cache from a previous experiment, we were able to start from where this experiment left off. This proved useful, as experiments could take several days.

Another practical problem besides non-determinism is that an SSH server may produce a sequence of outputs in response to a single input. This means it is not behaving as a Mealy machines, which allows for only one output. To deal with this, the MAPPER concatenates all outputs into one, producing a single output which it delivers to the learner. The sequence in which the concatenated outputs are received is preserved in the produced output.

A final challenge is presented by forms of ‘buffering’, which we encountered in two situations. Firstly, some implementations buffer incoming requests during rekey; only once rekeying is complete are all these messages processed. This leads to a NEWKEYS

[3]The IGNORE messages are aimed to thwart traffic analysis.

response (indicating rekeying has completed), directly followed by all the responses to the buffered requests. This would lead to non-termination of the learning algorithm, as for every sequence of buffered messages the response differs. To prevent this, we treat the sequence of queued responses as the single output BUFFERED.

A different form of buffering occurs when opening and closing channels, since a SUL can close only as many channels as have previously been opened. Learning this behavior would lead to an infinite state machine, as we would need a state 'there are n channels open' for every number n. For this reason, we restrict the number of simultaneously open channels to one. The MAPPER returns a custom response CH_MAX to a CH_OPEN message whenever this limit is reached.

3.5 Learning Results

We use the setup described in Section 3.4 to learn models for OpenSSH, Bitvise and DropBear SSH server implementations. OpenSSH represents the focal point, as it is the most popular implementation of SSH (with over 80 percent of market share in 2008 [18]) and the default server for many UNIX-based systems. DropBear is an alternative to OpenSSH designed for low resource systems. Bitvise is a well-known proprietary Windows-only SSH implementation.

In our experimental setup, learner and MAPPER ran inside a Linux Virtual Machine. OpenSSH and DropBear were learned over a localhost connection, whereas Bitvise was learned over a virtual connection with the Windows host machine. We have adapted the setting of timing parameters to each implementation.

OpenSSH was learned using a full alphabet, whereas DropBear and Bitvise were learned using a restricted alphabet (as defined in Subsection 3.4.1). The reason for using a restricted alphabet was to reduce learning times. Based on the model learned for OpenSSH (the first implementation analyzed) and the specification, we excluded inputs that seemed unlikely to produce state change (such as DEBUG or UNIMPL). We also excluded inputs that could take a long time to process (such as DISCONNECT) but were not were not needed to visit all states in the happy flow. We excluded, for example, the user/password based authentication inputs (UA_PW_OK and UA_PW_NOK) as they would take the system 2-3 seconds to respond to. By contrast, public key authentication resulted in quick responses. We explain the more intricate case for DISCONNECT when we discuss the learned models.

For generating test queries we used random and exhaustive variants of the testing algorithm described in [191], which generate efficient test suites. Tests generated comprise an access sequence, a middle section of length k and a distinguishing sequence. The exhaustive variant generates tests for all possible middle sections of length k and all states. Passing all tests then provides some notion of confidence, namely, that the learned model is correct unless the (unknown) model of the implementation has at least k more states than the learned hypothesis. The random variant produces tests with randomly generated middle sections. No formal confidence is provided, but

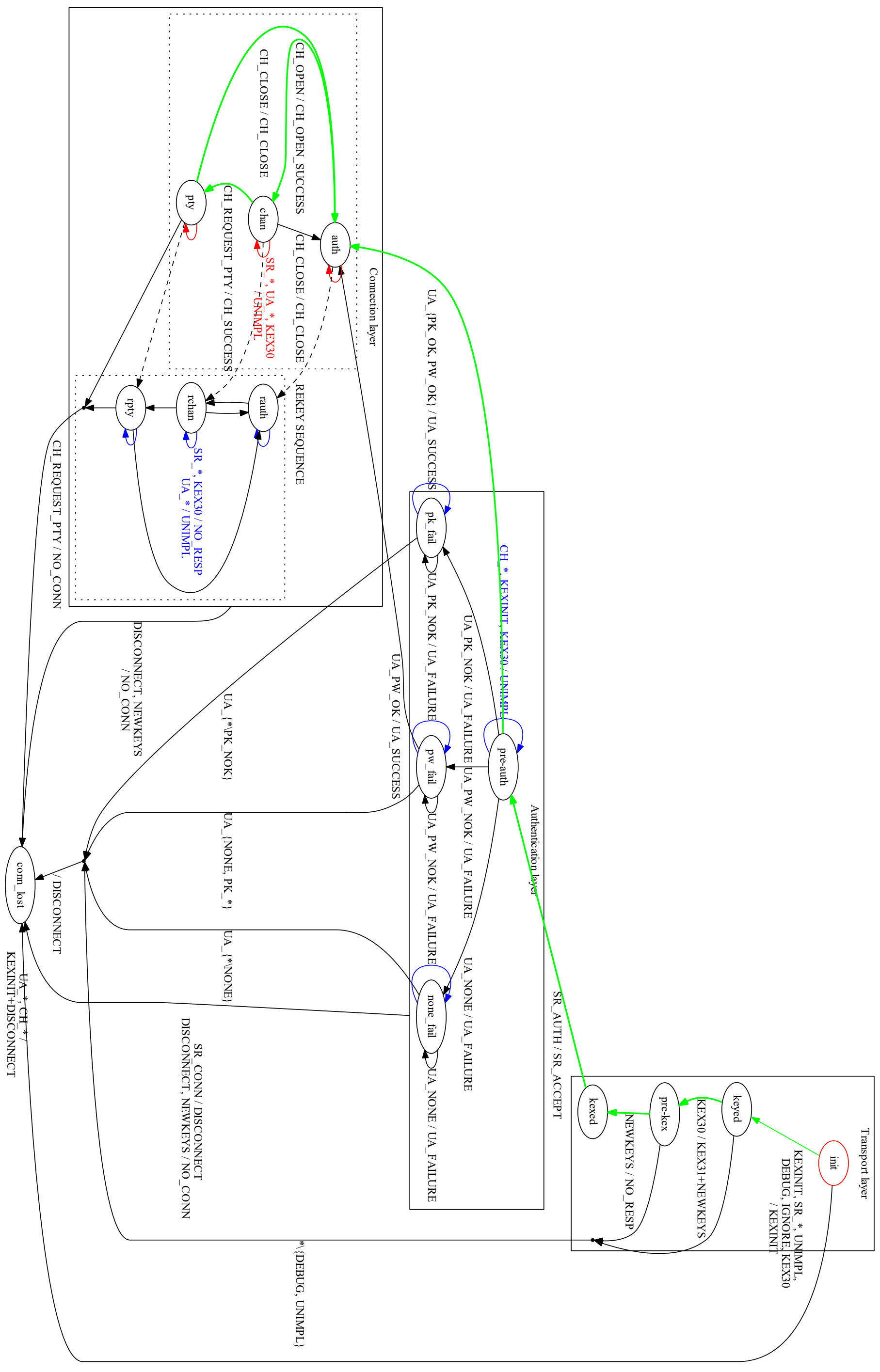


Figure 3.6: Model of the OpenSSH server.

past experience shows this to be more effective at finding counterexamples since k can be set to higher values. We executed a random test suite with k of 4 comprising 40000 tests for OpenSSH, and 20000 tests for Bitvise and DropBear. We then ran an exhaustive test suite with k of 2 for all implementations.

Extending our test suite are test queries derived manually from counterexamples generated by model checking the learned model. These counterexamples have to be checked on the SUL to see whether the SUL is indeed non-compliant with the corresponding property or the learned model is wrong. As the properties we formalized were safety properties, all resulting counterexamples were of finite length. We hence were able to manually derive tests from them, and add these tests to the test suite. In the case of DropBear, one of the counterexamples found by the model checker was an actual counterexample for the learner (it invalidated the learned model). This counterexample could not be found by our exhaustive test algorithm using a k of 2. Indeed, the hypothesis the learner produced after processing this counterexample had two additional states, indicating that a k of 3 would have been necessary. Such a test setting would have required the costly execution of 122836 tests on the invalidated learned model and 209911 on the final model. This shows the limitation of exhaustive test algorithms in finding counterexamples. It also provides a compelling argument for integrating a model checker into the testing loop for each hypothesis, rather than only for the learned model.

Table 3.4 describes the exact versions of the systems analyzed together with statistics on learning and testing: (1) the number of states in the learned model, (2) the number of hypotheses built during the learning process and (3) the total number of learning and test queries run. For test queries, we only consider those run on the last hypothesis. All learned models are available at [4]. Statistics give a glimpse into the issue of scalability. Assuming each input took 0.5 seconds to process, and an average query length of 10, to perform 40000 queries would have taken roughly 55 hours. This is consistent with the time experiments took, which span several days. The long duration compelled us to resort to restricted alphabets, which lead to reduction in the number of queries needed. Our work could have benefited from parallel execution.

Table 3.4: Statistics for learning experiments

SUT	**States**	**Hypotheses**	**Mem. Q.**	**Test Q.**
OpenSSH 6.9p1-2	31	4	19836	76418
Bitvise 7.23	65	15	24996	58423
DropBear v2014.65	29	8	8357	64478

The large number of states is down to several reasons. First of all, some systems exhibited buffering behavior. In particular, Bitvise would queue responses for higher layer inputs sent during rekey, and would deliver them all at once after rekeying was done. Interestingly, the size of the queue affected how Bitvise reacted to DISCONNECT

[4] https://gitlab.science.ru.nl/pfiteraubrostean/Learning-SSH-Paper/tree/master/models

during rekey, a lower layer input. The longer the queue, the more time it took for BitVise to process and thus, terminate the connection. This forced us to exclude DISCONNECT from the restricted alphabet, as its processing time could grow to several seconds. Rekeying was another major contributor to the number of states. For each state where rekeying is possible, the sequence of transitions constituting the complete rekeying process should lead back to that state. This leads to two additional rekeying states for every state allowing rekey. Many states were also added due to the MAPPER generated outputs CH_NONE or CH_MAX, outputs which signal that no channel is open or that the maximum number of channels have been opened.

Figure 3.6 shows the model obtained for OpenSSH, with some edits to improve readability. The figure collects the states into 3 clusters, indicated by the rectangles, where each cluster corresponds to one of the protocol layers. We eliminate redundant states and information induced by the MAPPER, as well as states present in successful rekeying sequences. Wherever rekeying was permitted, we replace the rekeying states and transitions by a single *REKEY SEQUENCE* transition. We also factor out edges common to states within a cluster. We replace common disconnecting edges, by one edge from the cluster to the disconnect state. Common self loop edges are colored, and the actual i/o information only appears on one edge. Transitions with similar start and end states are joined together on the same edge. Transition labels are kept short by regular expressions(UA_* stands for inputs starting with UA_) or by factoring out common start strings. Green edges highlight the happy flow. '+' concatenates multiple outputs.

On analyzing Figure 3.6, we notice that the happy flow, colored in green, is fully explored and mostly matches our earlier description of it[5]. Also explored is what happens when a rekeying sequence is attempted. We notice that rekeying is only allowed in states of the Connection layer. Strangely, for these states, rekeying is not state preserving, as the generated output on receiving a SR_AUTH, SR_CONN or KEX30 changes from UNIMPL to NO_RESP. This leads to two sub-clusters of states, one before the first rekey, the other afterward. In all other states, the first step of a rekeying (KEXINIT) yields (UNIMPL), while the last step (NEWKEYS) causes the system to disconnect.

We also note the intricate authentication behavior: after an unsuccessful authentication attempt the only authentication method still allowed is password authentication. Finally, only Bitvise allowed multiple terminals to be requested over the same channel. As depicted in the model, OpenSSH abruptly terminates on requesting a second terminal. DropBear exhibits a similar behavior.

We warn the reader of an inaccuracy in the learned models caused by the MAPPER. The inaccuracy was detected after experiments were done and is still present. Analyzing the initial state, we remark how KEXINIT appears as response to most inputs from that

[5]The only exception is in the Transport layer, where unlike in our happy flow definition, the server is the first to send the NEWKEYS message. This is also accepted behavior, as the protocol does not specify which side should send NEWKEYS first.

state. This behavior is odd, since we expected KEXINIT responses to be given only to KEXINIT messages. Upon closer inspection, we found that a KEXINIT message was sent by the server just after SSH versions had been exchanged, before the server received any input from the learner. Our MAPPER buffered this KEXINIT message and falsely considered it as a response to the first input sent by the learner. Consequently, KEXINIT appears as response to all inputs in the initial state, though it was actually generated before any input was sent. Furthermore, when parsing this KEXINIT response, the MAPPER could silently send KEXINIT to the server effectively completing preference exchange. This happened if the input was not KEXINIT and did not cause the connection to terminate. Later processing of KEXINIT messages is done normally (such as during rekey). Readers should keep this inaccuracy in mind when interpreting the model of Figure 3.6. The inaccuracy affects solely transitions from the start state. In the future, we hope to address this inaccuracy, reduce MAPPER-induced redundant states and update [4] with improved models

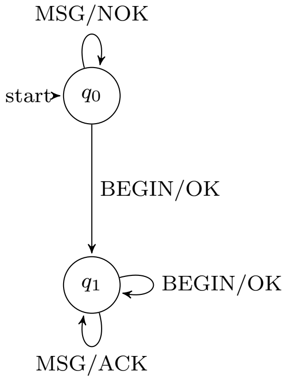
3.6 Security Specifications

A NuSMV model is specified by a set of finite variables together with a transition-function that describes changes on these variables. Specifications in temporal logic, such as CTL and LTL, can be checked for truth on specified models. NuSMV provides a counterexample if a given specification is not true. We generate NuSMV models automatically from the learned models. Generation proceeds by first defining a NuSMV file with three variables, corresponding to inputs, outputs and states. The transition-function is then extracted from the learned model and appended to this file. This function updates the output and state variables for a given valuation of the input variable and the current state. Figure 3.7 gives an example of a Mealy machine and its associated NuSMV model. The NuSMV models derived from the learned SSH models, and the formalized properties are available at [4].

The remainder of this section defines the properties we formalized and verified. We group these properties into four categories:

1. *basic characterizing properties*, properties which characterize the MAPPER and SUL assembly at a basic level. These hold for all implementations.
2. *security properties*, these are properties fundamental to achieving the main security goal of the respective layer.
3. *key re-exchange properties*, or properties regarding the rekey operation (after the initial key exchange was done).
4. *functional properties*, which are extracted from the SHOULD's and the MUST's of the RFC specifications. They may have a security impact.

A key point to note is that properties are checked not on the actual concrete model of the SUL, but on an abstraction of the SUL that is induced by the MAPPER. This is unlike in [88], where properties were checked on a concretization of the learned



```
MODULE main
  VAR state : {q0, q1};
  inp : {BEGIN, MSG};
  out : {OK, NOK, ACK};
  ASSIGN
  init(state) := q0;
  next(state) := case
    state = q0 & inp = BEGIN: q1;
    state = q0 & inp = MSG: q0;
    state = q1 & inp = BEGIN: q1;
    state = q1 & inp = MSG: q1;
    esac;
  out := case
    state = q0 & inp = BEGIN: OK;
    state = q0 & inp = MSG: NOK;
    state = q1 & inp = BEGIN: OK;
    state = q1 & inp = MSG: ACK;
  esac;
```

Figure 3.7: Mealy machine + associated NuSMV code

model obtained by application of a reverse mapping. Building a reverse mapper is far from trivial given the MAPPER's complexity. Thus we need to be careful when we interpret model checking results for the learned model. Also, we must be aware that when some property does not hold for the abstract model, and the model checker provides a counterexample, we still need to check whether this counterexample is an actual run of the abstraction of the SUL induced by the MAPPER. If this is not the case then the counterexample demonstrates that the learned model is incorrect. In the previous section we noted an inconsistency between learned models and how the systems behave due to MAPPER-induced behavior. The inconsistency has minimal impact on the properties we analyze as it impacts only transitions from the initial state, and its effects are clear and can be accounted for.

Before introducing the properties, we mention some basic predicates and conventions we use in their definition. The happy flow in SSH consists in a series of steps: the user (1) exchanges keys, (2) requests for the authentication service, (3) supplies valid credentials to authenticate and finally (4) opens a channel. Whereas step (1) is complex, the subsequent steps can be captured by the simple predicates *hasReqAuth*, *validAuthReq* and *hasOpenedChannel* respectively. The predicates are defined in terms of the output generated at a given moment, with certain values of this output indicating that the step was performed successfully. For example, CH_OPEN_SUCCESS indicates that a channel has been opened successfully. Sometimes we also need the input that generated the output, so as to distinguish this step from other steps. In particular, requesting the authentication service is distinguished from requesting the connection service by SR_AUTH. To these predicates, we add predicates for valid, invalid and all authentication methods, a predicate for the receipt of NEWKEYS from the server, and receipt of KEXINIT, which can also be seen as initiation of key (re-) exchange. These last predicates have to be tweaked in accordance with the input alphabet used and with the output the SUL generated (KEXINIT could be sent in different packaging, either alone, or joined by a different message). Their formulations below are for the OpenSSH server. Finally, *connLost* indicates that connection was lost, and *endCondition* is the condition after which higher layer properties no longer have to hold.

```
hasReqAuth := inp=SR_AUTH ∧ out=SR_ACCEPT;
validAuthReq := out=UA_PK_OK ∨ out=UA_PW_OK;
hasOpenedChannel := out=CH_OPEN_SUCCESS;
validAuthReq := inp=UA_PK_OK ∨ inp=UA_PW_OK;
invAuthReq := inp=UA_PK_NOK ∨ inp=UA_PW_NOK ∨ inp=UA_NONE;
authReq := validAuthReq ∨ invalidAuthReq;
receivedNewKeys := out=NEWKEYS ∨ out=KEX31_NEWKEYS;
kexStarted := out=KEXINIT;
connLost := out=NO_CONN ∨ out=DISCONNECT;
endCondition := kexStarted ∨ connLost;
```

Our formulation uses NuSMV syntax. We also use the weak until operator W, which is not supported by NuSMV, but can be easily defined in terms of the until operator U and globally operator G that are supported: $p\,W\,q\; =\; p\,U\,q\,|\,G\,p$. Many of the higher layer properties we formulate should hold only until a disconnect or a key (re-)exchange happens, hence the definition of the *endCondition* predicate. This is because the RFCs don't specify what should happen when no connection exists. Moreover, higher layer properties in the RFCs only apply outside of rekey sequences, as inside a rekey sequence the RFCs advise implementations to reject all higher layer inputs, regardless of the state before the rekey.

3.6.1 Basic Characterizing Properties

In our setting, a single TCP connection is made and once this connection is lost (e.g. because the system disconnects) it cannot be re-established. The moment a connection is lost is marked by generation of the NO_CONN output. From this moment onwards, the only outputs encountered are the NO_CONN output (the MAPPER tried but failed to communicate with the SUL), or outputs generated by the MAPPER directly, without querying the system. The latter are CH_MAX (channel buffer is full) and CH_NONE (channel buffer is empty). With these outputs we define Property 3.1 which describes the "one connection" property of our setup.

Property 3.1.
```
G (out=NO_CONN →
    G (out=NO_CONN ∨ out=CH_MAX ∨ out=CH_NONE) )
```

Outputs CH_MAX and CH_NONE are still generated because of a characteristic we touched on in Subsection 3.4.2. The MAPPER maintains a buffer of open channels and limits its size to 1. From the perspective of the MAPPER, a channel is open, and thus added to the buffer, whenever CH_OPEN is received from the learner, regardless if a channel was actually opened on the SUL. In particular, if after opening a channel via CH_OPEN an additional attempt to open a channel is made, the MAPPER itself responds by CH_MAX without querying the SUL. This continues until the learner closes the channel by CH_CLOSE, prompting removal of the channel and the sending of an actual CLOSE message to the SUL (hence out!=CH_NONE). A converse property can be formulated in a similar way for when the buffer is empty after a CH_CLOSE, in which case subsequent CH_CLOSE messages prompt the MAPPER generated CH_NONE, until a channel is opened via CH_OPEN and an actual OPEN message is sent to the SUL. Conjunction of these two behaviors forms Property 3.2.

Property 3.2.
```
(G (inp=CH_OPEN) →
        X ( (inp=CH_OPEN → out=CH_MAX)
           W (inp=CH_CLOSE ∧ out!=CH_NONE) ) ) ∧
   (G (inp=CH_CLOSE) →
        X ( (inp=CH_CLOSE → out=CH_NONE)
           W (inp=CH_OPEN ∧ out!=CH_MAX) ) )
```

3.6.2 Security Properties

In SSH, upper layer services rely on security guarantees ensured by lower layers. So these services should not be available before the lower layers have completed. For example, the authentication service should only become available *after* a successful key exchange and the setting up of a secure tunnel by the Transport layer, otherwise the service would be running over an unencrypted channel. Requests for this service should therefore not succeed unless key exchange was performed successfully.

Key exchange involves three steps that have to be performed in order but may be interleaved by other actions. Successful authentication necessarily implies successful execution of the key exchange steps. We can tell each key exchange step was successful from the values of the input and output variables. Successful authentication request is indicated by the predicate defined earlier, *hasReqAuth*. Following these principles, we define the LTL specification in Property 3.3, where O is the once operator. Formula *Op* is true at time t if p held in at least one of the previous time steps $t' \leq t$.

Property 3.3.
```
G ( hasReqAuth →
      O ( (inp=NEWKEYS ∧ out=NO_RESP) ∧
         O ( (inp=KEX30 ∧ out=KEX31_NEWKEYS) ∧
            O (out=KEXINIT) ) ) )
```

Apart from a secure connection, Connection layer services also assume that the client behind the connection was authenticated. This is ensured by the Authentication layer by means of an authentication mechanism, which only succeeds, and thus authenticates the client, if valid credentials are provided. For the implementation to be secure, there should be no path from an unauthenticated to an authenticated state without the provision of valid credentials. We consider an authenticated state as a state where a channel has been opened successfully, described by the predicate *hasOpenedChannel*. Provision of valid/invalid credentials is indicated by the outputs UA_SUCCESS and UA_FAILURE respectively. Along these lines, we formulate this specification by Property 3.4, where S stands for the since operator. Formula pSq is true at time t if q held at some time $t' \leq t$ and p held in all times t'' such that $t' < t'' \leq t$.

Property 3.4. $\boldsymbol{G}$ (*hasOpenedChannel* → *out!=UA_FAILURE* $\boldsymbol{S}$ *out=UA_SUCCESS*)

3.6.3 Key Re-exchange Properties

According to the RFC [222, p. 24], re-exchanging keys (or rekeying) (1) is preferably allowed in all states of the protocol, and (2) its successful execution does not affect operation of the higher layers. We consider two general protocol states, pre-authenticated (after a successful authentication request, before authentication) and authenticated. These may map to multiple states in the learned models. We formalized requirement (1) by two properties, one for each general state. In the case of the pre-authenticated state, we know we have reached this state following a successful authentication service request, indicated by the predicate *hasReqAuth*. Once here, performing the inputs for rekey in succession should imply success until one of two things happen, the connection is lost(*connLost*) or we have authenticated. This is asserted in Property 3.5. A similar property is defined for the authenticated state.

Property 3.5. $\boldsymbol{G}$ (*hasReqAuth* →
  $\boldsymbol{X}$ (*inp=KEXINIT* → *out=KEXINIT* ∧
    $\boldsymbol{X}$ (*inp=KEX30* → *out=KEX31_NEWKEYS* ∧
      $\boldsymbol{X}$ (*inp=NEWKEYS* → *out=NO_RESP*))) $\boldsymbol{W}$
        (*connLost* ∨ *hasAuth*))

Requirement (2) cannot be expressed in LTL, since in LTL we cannot specify that two states are equivalent. We therefore checked this requirement directly, by writing a simple script which, for each state q that allows rekeying, checks if the state q' reached after a successful rekey is equivalent to q in the subautomaton that only contains the higher layer inputs.

3.6.4 Functional Properties

We formalized and checked several other properties drawn from the RFCs. We found parts of the specification unclear, which sometimes meant that we had to give our own interpretation. A first general property can be defined for the DISCONNECT output. The RFC specifies that after sending this message, a party MUST not send or receive any data [224, p. 24]. While we cannot tell what the server actually receives, we can check that the server does not generate any output after sending DISCONNECT. After a DISCONNECT message, subsequent outputs should be solely derived by the MAPPER.

Knowing the MAPPER induced outputs are NO_CONN, CH_MAX and CH_NONE, we formulate by Property 3.6 to describe expected outputs after a DISCONNECT.

Property 3.6. ***G*** *(out=DISCONNECT →*
X G *(out=CH_NONE ∨ out=CH_MAX ∨ out=NO_CONN))*

The RFC states in [222, p. 24] that after sending a KEXINIT message, a party MUST not send another KEXINIT, or a SR_ACCEPT message, until it has sent a NEWKEYS message($receivedNewKeys$). This is translated to Property 3.7.

Property 3.7. ***G*** *(out=KEXINIT →*
X *((out!=SR_ACCEPT ∧ out!=KEXINIT)* ***W*** *receivedNewKeys))*

The RFC also states [222, p. 24] that if the server rejects the service request, "it SHOULD send an appropriate SSH_MSG_DISCONNECT message and MUST disconnect". Moreover, in case it supports the service request, it MUST send a SR_ACCEPT message. Unfortunately, it is not evident from the specification if rejection and support are the only allowed outcomes. We assume that is the case, and formalize an LTL formula accordingly by Property 3.8. For a service request (SR_AUTH), in case we are not in the initial state, the response will be either an accept (SR_ACCEPT), disconnect (DISCONNECT), or NO_CONN, output generated by the MAPPER after the connection is lost. We adjusted the property for the initial state in which models responded with KEXINIT which would easily break the property. As explained in Section 3.5, systems actually generate this KEXINIT message before any input is sent, whereas models falsely encode it as a response to initial inputs.

Property 3.8. ***G*** *((inp=SR_AUTH ∧ state!=s0) →*
(out=SR_ACCEPT ∨ out=DISCONNECT ∨ out=NO_CONN)))

The RFC for the Authentication layer states in [221, p. 6] that if the server rejects the authentication request, it MUST respond with a UA_FAILURE message. Rejected requests are suggested by the predicate *invAuthReq*. In case of requests with valid credentials (*validAuthReq*), a UA_SUCCESS MUST be sent only once. While not explicitly stated, we assume this to be in a context where the authentication service had been successfully requested, hence we use the *hasReqAuth* predicate. We define two properties, Property 3.9 for behavior before an UA_SUCCESS, Property 3.10 for behavior afterward. For the first property, note that (*hasReqAuth*) may hold even after successful authentication, but we are only interested in behavior between the first time (*hasReqAuth*) holds and the first time authentication is successful (out=UA_SUCCESS), hence the use of the O operator. As is the case with most higher layer properties, the first property only has to hold until the end condition holds ($endCondition$), that is the connection is lost (*connLost*) or rekey was started by the SUL ($kexStarted$).

Property 3.9. ***G*** *((hasReqAuth ∧ !**O** out=UA_SUCCESS) →*

```
(invalidAuthReq → out=UA_FAILURE)
  W (out=UA_SUCCESS ∨ endCondition) )
```

Property 3.10. *G (out=UA_SUCCESS → X G out!=UA_SUCCESS)*

In the same paragraph, it is stated that authentication requests received after a UA_SUCCESS SHOULD be ignored. This is a weaker statement, and it requires that all authentication messages (suggested by *authReq*) after a UA_SUCCESS output should prompt no response from the system(NO_RESP) until the end condition is true. The formulation of this statement shown in Property 3.11.

Property 3.11. *G (out=UA_SUCCESS →*
X ((authReq→ out=NO_RESP) W endCondition))

The Connection layer RFC states in [222, p. 9] that on receiving a CH_CLOSE message, a party MUST send back a CH_CLOSE, unless it had already sent this message for the channel. The channel must have been opened beforehand (*hasOpenedChannel*) and the property only has to hold until the end condition holds or the channel was closed (*out=CH_CLOSE*). We formulate Property 3.12 accordingly.

Property 3.12. *G (hasOpenedChannel →*
((inp=CH_CLOSE) → (out=CH_CLOSE))
W (endCondition ∨ out=CH_CLOSE))

3.6.5 Model Checking Results

Table 3.5 presents model checking results. Crucially, the security properties hold for all three implementations. We had to slightly adapt our properties for Bitvise as it buffered all responses during rekey (incl. UA_SUCCESS). In particular, we used *validAuthReq* instead of *out=UA_SUCCESS* as sign of successful authentication.

Properties marked with '*' did not hold because implementations chose to send UNIMPL, instead of the output suggested by the RFC. As an example, after successful authentication, both Bitvise and OpenSSH respond with UNIMPL to further authentication requests, instead of being silent, violating Property 3.11. Whether the alternative behavior adapted is acceptable is up for debate. Certainly the RFC does not suggest it, though it does leave room for interpretation.

DropBear is the only implementation that allows rekeying in both general states of the protocol. DropBear also satisfies all Transport and Authentication layer specifications, however, problematically, it violates the property of the Connection layer. Upon receiving CH_CLOSE, it responds by CH_EOF instead of CH_CLOSE, not respecting Property 3.12.

Table 3.5: Model checking results

	Property	Key word	OpenSSH	Bitvise	DropBear
Security	Trans.		✓	✓	✓
	Auth.		✓	✓	✓
Rekey	Pre-auth.		X	✓	✓
	Auth.		✓	X	✓
Functional	Prop. 3.6	MUST	✓	✓	✓
	Prop. 3.7	MUST	✓	✓	✓
	Prop. 3.8	MUST	X*	X	✓
	Prop. 3.9	MUST	✓	✓	✓
	Prop. 3.10	MUST	✓	✓	✓
	Prop. 3.11	SHOULD	X*	X*	✓
	Prop. 3.12	MUST	✓	✓	X

3.7 Conclusions

We have combined model learning with abstraction techniques to infer models of the OpenSSH, Bitvise and DropBear SSH server implementations. We have also formalized several security and functional properties drawn from the SSH RFC specifications. We have verified these properties on the learned models using model checking and have uncovered several minor standard violations. The security-critical properties were met by all implementations.

Abstraction was provided by a MAPPER component placed between the learner and the SUL. The MAPPER was constructed from an existing SSH implementation. The input alphabet of the MAPPER explored key exchange, setting up a secure connection, several authentication methods, and opening and closing channels over which the terminal service could be requested. We used two input alphabets, a full version for OpenSSH, and a restricted version for Bitvise and DropBear. The restricted alphabet was still sufficient to explore most aforementioned behavior.

We encountered several challenges. Firstly, building a MAPPER presented a considerable technical challenge, as it required re-structuring of an actual SSH implementation. Secondly, because we used classical learning algorithms, we had to ensure that the abstracted implementation behaved like a (deterministic) Mealy machine. Here time-induced non-determinism was difficult to eliminate. Buffering also presented problems, leading to a considerable increase in the number of states. Moreover, the systems analyzed were relatively slow, which meant learning took several days. This was compounded by the size of the learning alphabet, and it forced us into using a reduced alphabet for two of the implementations.

Limitations of work in this chapter, hence possibilities for future work, are several. First of all, the MAPPER was not formalized, unlike in [88], thus we did not produce a concretization of the abstract models. Consequently, model checking results cannot be fully transferred to the actual implementations. Formal definition of the mapper and concretization of the learned models (as defined in [10]) would tackle this. The MAPPER also caused considerable redundancy in the learned models; tweaking the abstractions used, in particular those for managing channels, could alleviate this problem while also improving learning times. This in turn would facilitate learning using expanded alphabets instead of resorting to restricted alphabets. Furthermore, the MAPPER abstraction could be refined, to give more insight into the implementations. In particular, parameters such as the session identifier could be extracted from the MAPPER and potentially handled by existing Register Automata learners [5, 54]. These learners can infer systems with parameterized alphabets, state variables and simple operations on data. Finally, we suppressed all timing-related behavior, as it could not be handled by the classical learners used; there is preliminary work on learning timed automata [98] which could use timing behavior.

Despite these limitations, work in this chapter provides a compelling application of learning and model checking in a security setting, on a widely used protocol. We hope this lays some more groundwork for further case studies, as well as advances in learning techniques.

Chapter 4

Learning Register Automata with Fresh Value Generation

We present a new algorithm for active learning of register automata. Our algorithm uses counterexample-guided abstraction refinement to automatically construct a component which maps (in a history dependent manner) the large set of actions of an implementation into a small set of actions that can be handled by a Mealy machine learner.

The class of register automata that is handled by our algorithm extends previous definitions since it allows for the generation of fresh output values. This feature is crucial in many real-world systems (e.g. servers that generate identifiers, passwords or sequence numbers). We have implemented our new algorithm in a tool called Tomte.

4.1 Introduction

Model checking and model learning are two core techniques in model-driven engineering. In model checking [68] one explores the state space of a given state transition model, whereas in model learning [19, 108, 198, 205] the goal is to obtain such a model through interaction with a system by providing inputs and observing outputs. Both techniques face a combinatorial blow up of the state-space, commonly known as the state explosion problem. In order to find new techniques to combat this problem, it makes sense to follow a cyclic research methodology in which tools are applied to challenging applications, the experience gained during this work is used to generate new theory and algorithms, which in turn are used to further improve the tools. After consistent application of this methodology for 25 years model checking is now applied routinely to industrial problems [1]. Work on the use of model learning in model-driven engineering started later [170] and has not yet reached the same maturity level, but in recent years there has been spectacular progress.

We have seen, for instance, several convincing applications of model learning in the area of security and network protocols. Cho et al. [64] successfully used model learning to infer models of communication protocols used by botnets. Model learning was used for fingerprinting of EMV banking cards [3]. It also revealed a security vulnerability in a smartcard reader for internet banking that was previously discovered by manual analysis, and confirmed the absence of this flaw in an updated version of this device [59]. Fiterau et al. [87] used model learning to demonstrate that both Linux and Windows implementations violate the TCP protocol standard. Using a similar approach, Fiterau et al. [89] showed that three implementations of the Secure Shell (SSH) protocol violate the standard. In [177], model learning is used to infer properties of a network router, and for testing the security of a web-application (the Mantis bug-tracker). Model learning has proven to be an extremely effective technique for spotting bugs, complementary to existing methods for software analysis.

A major theoretical challenge is to lift learning algorithms for finite state systems to richer classes of models involving data. A breakthrough has been the definition of a Nerode congruence for a class of register automata [55, 56] and the resulting generalization of learning algorithms to this class [114,115]. Register automata [55,126] are a type of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Recently, the results on register automata have been generalized to even larger classes of models in which guards may contain arithmetic constraints and inequalities [54, 58]. A different approach for extending learning algorithms to classes of models involving data has been proposed in [10]. Here the idea is to place an intermediate mapper component in between the implementation and the learner. This mapper abstracts (in a history dependent manner) the large set of (parametrized) actions of the implementation into a small set of abstract actions that can now be handled by automata learning algorithms for finite state systems. In [7], we described an algorithm that uses counterexample-guided abstraction refinement to automatically construct an appropriate mapper for a subclass of register automata that may only store the first and the last occurrence of a parameter value. Moerman et al. [156] present a learning algorithm for *nominal automata*, which are acceptors of languages over infinite (structured) alphabets. Nominal automata are a direct reformulation of the classical notion of finite automaton where one replaces finite sets with orbit-finite sets and functions (or relations) with equivariant ones [35, 172]. The algorithm of Moerman et al. [156] is almost a verbatim copy of the classical algorithm of Angluin [19]. Deterministic nominal automata are equally expressive as register automata but, due to the fact that they are unique-valued, exponentially less succinct (see [56]).

The approaches of [7, 54–56, 58, 114, 115, 156] do not allow to learn models with of fresh output values. Fresh outputs are technically challenging,[1] but crucial in many

[1]In register automata frameworks with accepting states, like [56], fresh outputs can be modeled but membership queries cannot be implemented, whereas in transducer based frameworks that generate outputs in response to inputs, like [7,114], the outcome of membership queries would become nondeterministic.

real-world systems, e.g. servers that generate fresh identifiers, passwords or sequence numbers. Bollig et al. [37] provide a learning algorithm for *session automata*, a class of register automata that supports fresh data values. This algorithm is elegant, but the expressivity of session automata is limited since no guards are allowed in transitions.

The main contributions of this chapter are (a) an extension of the learning algorithm of [7] to the full class of register automate with fresh outputs, (b) a description of the implementation of our new algorithm in the 0.41 release of the Tomte tool[2], and (c) an experimental evaluation of our implementation on a series of benchmarks, including a comparison with the RALib [54] tool. Tomte's source code and experimental results are available at[3].

Figure 4.1 presents the overall architecture of our learning approach. At the right we see the *teacher* or *system under learning (SUL)*, an implementation whose behavior can be described by an (unknown) input enabled and input deterministic register automaton. At the left we see the *learner*, which is a tool for learning finite deterministic Mealy machines. In our current implementation we use LearnLib [154, 178], but there are also other libraries like libalf [38] that implement active learning algorithms. In between the learner and the SUL we place three auxiliary components: the *determinizer*, the *lookahead oracle*, and the *abstractor*. First the determinizer eliminates the nondeterminism of the SUL that is induced by fresh outputs. Then the lookahead oracle annotates events with information about the data values that need to be remembered because they play a role in the future behavior of the SUL. Finally, the abstractor maps the large set of concrete values of the SUL to a small set of symbolic values that can be handled by the learner.

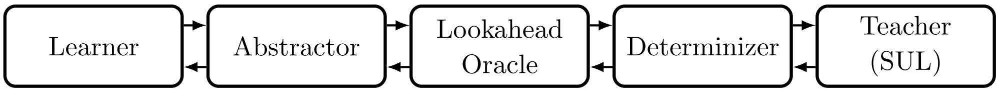


Figure 4.1: Architecture of Tomte

The idea to use an abstractor for learning register automata originates from [7] (based on work of [10]). Using abstractors one can only learn restricted types of deterministic register automata. Therefore, [2,8] introduced the concept of a lookahead oracle, which makes it possible to learn any deterministic register automaton. In this chapter, we extend the algorithm of [2, 8] with the notion of a determinizer, allowing us to also learn register automata with fresh outputs.

[2]http://tomte.cs.ru.nl/

[3]https://gitlab.science.ru.nl/harcok/tomte/tree/release-0.41

4.2 Register Automata

In this section, we define register automata and their operational semantics in terms of Mealy machines. In addition, we discuss technical concepts that provide insight into the behavior of register automata, concepts which play a key role in the technical development of this chapter: right invariance and symmetry. For reasons of exposition, the notion of *register automaton* that we define here is a simplified version of what we have implemented in our tool: Tomte also supports constants and actions with multiple parameters. Our definition is similar to that of a *scalarset Mealy machine* from [7] and of a *register Mealy machine* from [114], except that the value of the output parameter is not necessarily determined by the values of the registers and the input parameter.

4.2.1 Definition

We postulate a countably infinite set $\mathcal{V}$ of *variables*, which contains two special variables in and out. An *atomic formula* is a boolean expression of the form true, false, $x = y$ or $x \neq y$, with $x, y \in \mathcal{V}$. A *formula* φ is a conjunction of atomic formulas. Let $X \subseteq \mathcal{V}$ be a set of variables. We write $\Phi(X)$ for the set of formulas with variables taken from X. A *valuation* for X is a function $\xi : X \to \mathbb{Z}$. We write $\mathsf{Val}(X)$ for the set of valuations for X. If φ is a formula with variables from X and ξ is a valuation for X, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. We use symbol $\equiv$ to denote syntactic equality of formulas. We represent (partial) functions as sets of pairs, and write $X \rightharpoonup Y$ for the set of partial functions from X to Y.

Definition 4.1 (Register automaton). *A* register automaton (RA) *is a tuple* $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ *with*

- I *and* O *finite, disjoint sets of* input *and* output symbols*, respectively,*
- L *a finite set of* locations *and* $l_0 \in L$ *the* initial location*,*
- V *is a function that assigns to each location* l *a finite set* $V(l) \subseteq \mathcal{V} \setminus \{\mathit{in}, \mathit{out}\}$ *of* registers*, with* $V(l_0) = \emptyset$*.*
- $\Gamma \subseteq L \times I \times \Phi(\mathcal{V}) \times (\mathcal{V} \rightharpoonup \mathcal{V}) \times O \times L$ *a finite set of* transitions*. For each transition* $\langle l, i, g, \varrho, o, l' \rangle \in \Gamma$*, we refer to* l *as the* source*,* i *as the input symbol,* g *as the* guard*,* ϱ *as the* update*,* o *as the* output symbol*, and* l' *as the* target*. We require that* $g \in \Phi(V(l) \cup \{\mathit{in}, \mathit{out}\})$ *and* $\varrho : V(l') \to V(l) \cup \{\mathit{in}, \mathit{out}\}$*. We write* $l \xrightarrow{i,g,\varrho,o} l'$ *if* $\langle l, i, g, \varrho, o, l' \rangle \in \Gamma$*.*

Example 4.1. *As a first running example of a register automaton we use a FIFO-set with capacity two, similar to the one presented in [114]. A FIFO-set is a queue in which only different values can be stored, see Figure 4.2. In this automaton,* $I = \{\mathit{Push}, \mathit{Pop}\}$*,* $O = \{\mathsf{OK}, \mathsf{NOK}, \mathit{Return}\}$*,* $L = \{l_0, l_1, l_2\}$*,* $V(l_0) = \emptyset$*,* $V(l_1) = \{v\}$*, and* $V(l_2) = \{v, w\}$*. Input* Push *tries to add the value of parameter* in *to the queue, and input* Pop *tries*

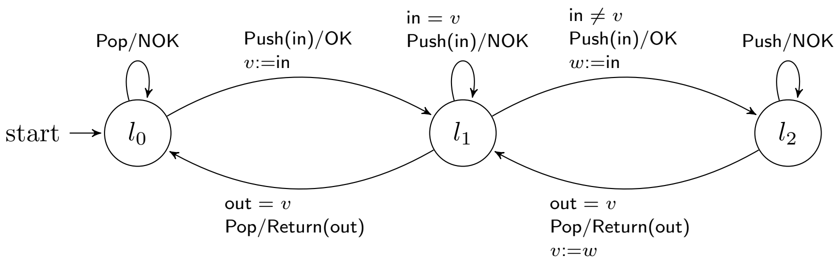


Figure 4.2: FIFO-set with a capacity of 2 modeled as a register automaton

to retrieve a value from the queue. The output in response to a Push *is* OK *if the input value can be added successfully, or* NOK *if the input value is already in the queue or if the queue is full. The output in response to a* Pop *is* Return(out)*, with as parameter the oldest value from the queue, or* NOK *if the queue is empty. Each input has parameter* in *and each output has parameter* out*. However, we omit parameters that do not matter and for instance write* Pop *instead of* Pop(in) *since parameter* in *does not occur in the guard and is not touched by the update. Usually, we also do not list the sets of variables of locations explicitly, as they can be inferred from the context.*

4.2.2 Semantics

The operational semantics of register automata is defined in terms of (infinite state) Mealy machines.

Definition 4.2 (Mealy machine)**.** *A* Mealy machine *is defined to be a tuple $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow\rangle$, where I and O are disjoint sets of* input *and* output actions*, respectively, Q is a set of* states*, $q^0 \in Q$ 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 state q' such that $q \xrightarrow{i/o} q'$. A Mealy machine is* input enabled *if, 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 sets Q, I and O are finite.*

A *partial run* of $\mathcal{M}$ is a finite sequence $\alpha = q_0\ i_0\ o_0\ q_1\ i_1\ o_1\ q_2 \cdots i_{n-1}\ o_{n-1}\ q_n$, beginning and ending with a state, such that for all $j < n$, $q_j \xrightarrow{i_j/o_j} q_{j+1}$. A *run* of $\mathcal{M}$ is a partial run that starts with q^0. The *trace* of α, denoted $\mathsf{trace}(\alpha)$, is the finite sequence $\beta = i_0\ o_0\ i_1\ o_1 \cdots i_{n-1}\ o_{n-1}$ that is obtained by erasing all the states from α. We say that β is a *trace* of state $q \in Q$ iff β is the trace of some partial run that starts in q, and we say that β is a *trace* of $\mathcal{M}$ iff β is a trace of q^0. We call two states $q, q' \in Q$ *equivalent*, notation $q \approx q'$, iff they have the same traces. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Mealy machines with the same sets of input actions. We say that $\mathcal{M}_1$ and $\mathcal{M}_2$ are *equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if they have the same traces. We say that $\mathcal{M}_1$ *implements* $\mathcal{M}_2$, notation $\mathcal{M}_1 \leq \mathcal{M}_2$, if all traces of $\mathcal{M}_1$ are also traces of $\mathcal{M}_2$.

The operational semantics of a register automaton is a Mealy machine in which the states are pairs of a location l and a valuation ξ of the state variables. A transition may fire for given input and output values if its guard evaluates to true. In this case, a new valuation of the state variables is computed using the update part of the transition.

Definition 4.3 (Semantics register automata)**.** *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma\rangle$ be a RA. The operational semantics of $\mathcal{R}$, denoted $[\![\mathcal{R}]\!]$, is the Mealy machine $\langle I \times \mathbb{Z}, O \times \mathbb{Z}, Q, q^0, \rightarrow\rangle$, where $Q = \{(l, \xi) \mid l \in L \wedge \xi \in \mathsf{Val}(V(l))\}$, $q^0 = (l_0, \emptyset)$, and relation $\rightarrow$ is defined inductively by the rule*

$$\frac{l \xrightarrow{i,g,\varrho,o} l' \qquad \iota = \xi \cup \{(\mathit{in}, d), (\mathit{out}, e)\} \quad \iota \models g \quad \xi' = \iota \circ \varrho}{(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')} \tag{4.1}$$

If transition $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ can be inferred using rule (4.1) then we say that it is supported *by transition $l \xrightarrow{i,g,\varrho,o} l'$ of $\mathcal{R}$, and that transition $l \xrightarrow{i,g,\varrho,o} l'$* fires*.*
We call $\mathcal{R}$ input enabled if its operational semantics $[\![\mathcal{R}]\!]$ is input enabled. A run or trace of $\mathcal{R}$ is just a run or trace of $[\![\mathcal{R}]\!]$, respectively. Two register automata $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent *if $[\![\mathcal{R}_1]\!]$ and $[\![\mathcal{R}_2]\!]$ are equivalent. We call $\mathcal{R}$* input deterministic *if for each reachable state (l, ξ) and input action $i(d)$ at most one transition may fire. An input deterministic register automaton $\mathcal{R}$ has the property that for any trace β of $\mathcal{R}$ there exists a unique run α such that $\mathsf{trace}(\alpha) = \beta$.*

Example 4.2. *The register automaton of Figure 4.2 is input deterministic. The following sequence constitutes a run of this automaton:*

$$(l_0, \emptyset) \xrightarrow{\mathit{Pop}/\mathsf{NOK}} (l_0, \emptyset) \xrightarrow{\mathit{Push}(22)/\mathsf{OK}} (l_1, \{(v, 22)\}) \xrightarrow{\mathit{Push}(7)/\mathsf{OK}} (l_1, \{(v, 22), (w, 7)\})$$

By erasing the states from this sequence we obtain the trace

$$\mathit{Pop}\ \mathsf{NOK}\ \mathit{Push}(22)\ \mathsf{OK}\ \mathit{Push}(7)\ \mathsf{OK}$$

Note that we again omit the parameters of actions Pop, NOK and OK.

The main contribution of this chapter is an algorithm for learning input enabled and input deterministic register automata. Our algorithm solves this problem by reducing it to the problem of learning finite *deterministic* Mealy machines, for which efficient algorithms exist. We recall the definition of a deterministic Mealy machine. We call a register automaton deterministic if its semantics is a deterministic Mealy machine.

Definition 4.4 (Deterministic Mealy machine)**.** *A Mealy machine $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$ is* deterministic *if for each state q and input action i there is exactly one output action 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 $\delta : Q \times I \to Q$ and $\lambda : Q \times I \to O$ are defined by: $q \xrightarrow{i/o} q' \;\Leftrightarrow\; \delta(q,i) = q' \wedge \lambda(q,i) = o$. 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 4.3. *The register automaton of Figure 4.2 is not deterministic. Recall that in every transition of a register automaton the input symbol carries a parameter* in *and the output symbol carries a parameter* out*, but that we omit these parameters in diagrams when they do not occur in the guard and are not touched by the update. As there are no constraints on the value of* out *for transitions with output symbol* OK*, an input* *Push*(1) *may induce both an* OK(1) *and an* OK(2) *output (in fact, parameter* out *can take any value). We can easily make the automaton of Figure 4.2 deterministic, for instance by strengthening the guards with* out = in *for transitions where the output value does not matter.*

Example 4.4. *Our second running example is a register automaton, displayed in Figure 4.3, that describes a simple login procedure. If a user performs a* Register*-input*

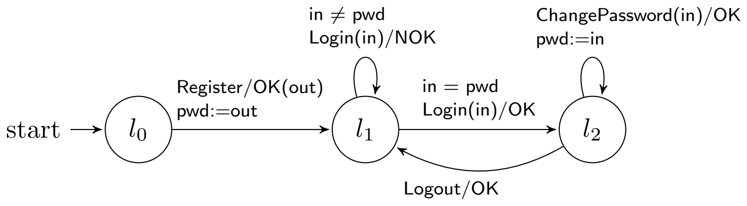


Figure 4.3: A simple login procedure modeled as a register automaton

then the automaton produces output symbol OK *together with a password. The user may then proceed by performing a* Login*-input together with the password that she has just received. After login the user may either change the password or logout. We can easily make the automaton input enabled by adding self loops i/*NOK *in each location, for each input symbol i that is not enabled. It is not possible to model the login procedure as a deterministic register automaton: the very essence of the protocol is that the system nondeterministically picks a password and gives it to the user.*

4.2.3 Symmetry

A key characteristic of register automata is that they exhibit strong symmetries. Because no operations on data values are allowed and we can only test for equality, bijective renaming of data values preserves behavior. The symmetries can be formally expressed through the notion of an automorphism. In the remainder of this section, we present the definition of an automorphism and explore some basic properties that will play a key role later on in this chapter. Slight variations of these properties have been proven elsewhere, see for instance [77]. The symmetry of register automata under data automorphisms plays a central role in [35], and in fact serves as a definition of the equivalent notion of a nominal automaton.

Definition 4.5. *An* automorphism *is a bijection $h : \mathbb{Z} \to \mathbb{Z}$.*

Let X be a set of variables. Then we lift an automorphism h to valuations ξ for X by pointwise extension, that is, $h(\xi) = h \circ \xi$. Since formulas in $\Phi(X)$ only assert that variables from X are equal or not, satisfaction of these formulas is not affected when we apply an automorphism to a valuation.

Lemma 4.1. *Let h be an automorphism, X be a set of variables, $\xi \in \mathsf{Val}(X)$ and $\varphi \in \Phi(X)$. Then $\xi \models \varphi$ iff $h(\xi) \models \varphi$.*

Proof. By structural induction on φ. □

We also lift automorphisms to the states, actions and transitions of a register automaton $\mathcal{R}$ by pointwise extension. The transition relation of $\mathcal{R}$ is preserved by automorphisms.

Lemma 4.2. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ is a transition of $\mathcal{R}$ iff $(l, h(\xi)) \xrightarrow{i(h(d))/o(h(e))} (l', h(\xi'))$ is a transition of $\mathcal{R}$.*

Proof. Use Lemma 4.1. □

Next, we lift automorphisms to runs by pointwise extension.

Lemma 4.3. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then α is a (partial) run of $\mathcal{R}$ iff $h(\alpha)$ is a (partial) run of $\mathcal{R}$.*

Proof. Use Lemma 4.2 and the fact that h trivially preserves the initial state. □

Finally, we lift automorphisms to traces by pointwise extension.

Lemma 4.4. *Let h be an automorphism, let $\mathcal{R}$ be a register automaton and let α be a partial run of $\mathcal{R}$. Then $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$.*

Proof. Use Lemma 4.3. □

Corollary 4.1. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then β is a trace of $\mathcal{R}$ iff $h(\beta)$ is a trace of $\mathcal{R}$.*

We call two states, actions, transitions, runs or traces *equivalent* if there exists an automorphism that maps one to the other.

4.2.4 Constants and Multiple Parameters

Tomte also supports constants and actions with multiple parameters. These features are convenient for modelling applications, but do not add any expressivity to the basic model of register automata.

Suppose $\mathcal{R}$ is a register automaton in which we would like to refer to distinct constants c_1 and c_2. Then we may extend $\mathcal{R}$ with the sequence of two transitions illustrated in Figure 4.4, starting from location l'_0 which is the initial location of the extended automaton. The first transition initializes c_1, which becomes a variable in our encoding,

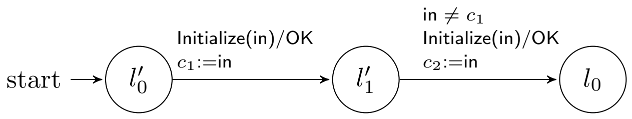


Figure 4.4: Encoding of constants

and similarly the second transition initializes c_2. After performing the initializations we enter the initial state l_0 of $\mathcal{R}$. Constants c_1 and c_2 are added as variables to all the locations of $\mathcal{R}$, and they may be tested in transitions. The encoding introduces an auxiliary input symbol Initialize and output symbol OK. If desired, the register automaton can be made input enabled by adding a trivial Initialize-loop to each location of $\mathcal{R}$, and an Initialize-loops with guard $\mathsf{in} = c_1$ to l'_1. Note that in an actual run of the automaton, c_1 and c_2 may be assigned arbitrary (distinct) values, different from the specific values for these constants that we had in mind originally. However, because of the symmetries of register automata this does not matter, and we may always rename constants to their intended values via an appropriate automorphism.

Tomte also supports multiple parameters for input and output actions, like in the simple login model shown in Figure 4.5. This model describes a system in which a user can register by providing a user id and a password, and then login using the credentials that were used for registering. What we can do here is to split a transition

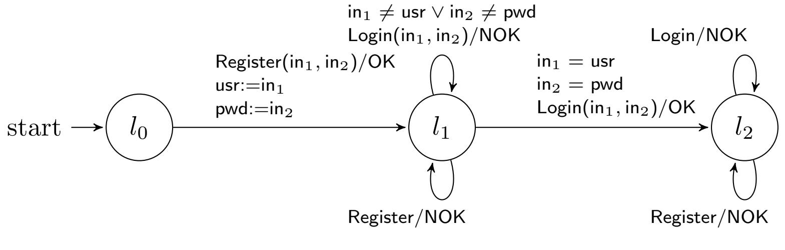

start
l_0
Register(in_1, in_2)/OK
usr:=in_1
pwd:=in_2
$\mathsf{in}_1 \neq$ usr $\vee$ $\mathsf{in}_2 \neq$ pwd
Login(in_1, in_2)/NOK
l_1
Register/NOK
in_1 = usr
in_2 = pwd
Login(in_1, in_2)/OK
Login/NOK
l_2
Register/NOK

Figure 4.5: A simple login system with inputs that carry two parameters

with multiple input parameters into a squence of transitions with a single parameter. The transition from l_0 to l_1, for instance, can be translated to the pattern shown in Figure 4.6.

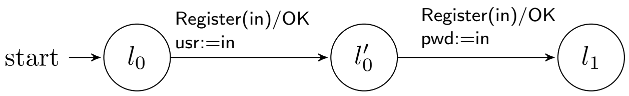


Figure 4.6: Encoding of multiple parameters

The implementation in Tomte involves several optimizations and does not use the above encodings. Nevertheless, these encodings show how constants and multiple parameters can be handled conceptually.

4.3 Restricted Types of Register Automata

Cassel et al [54] introduce the concept of a *right invariant* register automaton and provide a canonical automaton presentation of any language recognizable by a deterministic right invariant register automaton. The notion of right invariance plays an important role in our work as well. In this section, we discuss the formal definition of right invariance and prove some key results.

Definition 4.6. *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma\rangle$ be a register automaton. Then $\mathcal{R}$ is* right invariant *if, for each transition $l \xrightarrow{i,g,\varrho,o} l'$ in Γ, g is satisfiable and*

1. *for distinct $x, y \in V(l)$, neither $g \Rightarrow x = y$ nor $g \Rightarrow x \neq y$ is valid, and*
2. *the combined effect of guard g and assignment ϱ does not imply $x = y$ for distinct $x, y \in V(l')$ (note that inequalities may be implied).*[4]

Right invariance says that in guards we may compare input and output values with registers, but we are not allowed to test for (in)equality of distinct registers. Also, assignments may not copy the value of a single register in the source state to two distinct registers in the target state.

Example 4.5. *The FIFO-set model of Figure 4.2 and the login model of Figure 4.3 are right invariant. Figure 4.7 shows an example of a register automaton that is not right invariant. This automaton models a simple slot machine. By pressing a button a user may stop a spinning reel to reveal a value. If two consecutive values are equal then the user wins, otherwise he loses. The automaton is not right invariant, since in location l_2 we test for equality the registers v and w.*

The next lemma provides an equivalent characterization of right invariance.

[4]We can formalize this second condition by introducing a primed version x' of each variable x: for all pairs of distinct variables $x, y \in V(l')$ we require that the implication $g \wedge (\bigwedge_{z \in V(l')} z' = \rho(z)) \Rightarrow x' = y'$ is not valid.

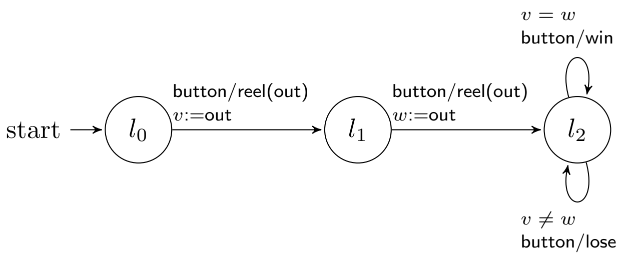


Figure 4.7: A simple slot machine modeled as a register automaton

Lemma 4.5. *$\mathcal{R}$ is right invariant iff for all transitions $l \xrightarrow{i,g,\varrho,o} l'$, ϱ is injective and guard g is equivalent to a formula of the form $g_{in} \wedge g_{out}$ such that*

1. *$g_{in} \equiv in = x$, with $x \in V(l)$, or $g_{in} \equiv \bigwedge_{x \in W} in \neq x$, with $W \subseteq V(l)$ (by convention the conjunction over the empty index set is* true*),*
2. *$g_{out} \equiv out = x$, with $x \in V(l) \cup \{in\}$, or $g_{out} \equiv \bigwedge_{x \in W} out \neq x$, with $W \subseteq V(l) \cup \{in\}$,*
3. *if $g_{in} \equiv in = x$, with $x \in V(l)$, then there are no $y, z \in V(l')$ with $\varrho(y) = in$ and $\varrho(z) = x$,*
4. *if $g_{out} \equiv out = x$, with $x \in V(l) \cup \{in\}$, then there are no $y, z \in V(l')$ with $\varrho(y) = out$ and $\varrho(z) = x$, and*
5. *there is no $x \in V(l)$ with $g_{in} \equiv in = x$ and $g_{out} \equiv out = x$.*

Proof. "$\Rightarrow$" Suppose $l \xrightarrow{i,g,\varrho,o} l'$ is a transition of $\mathcal{R}$. Assume that $x, y \in V(l')$ are distinct variables and $\varrho(x) = \varrho(y)$. Then $g \wedge (\bigwedge_{z \in V(l')} z' = \rho(z)) \Rightarrow x' = y'$ is valid, which is a contradiction. Thus ϱ is injective. Since g is satisfiable, it can be written as a conjunction of atomic formula $x = y$ or $x \neq y$ with x, y distinct variables from $V(l) \cup \{\mathsf{in}, \mathsf{out}\}$. W.l.o.g. we assume that each atomic formula occurs at most once in g. Observe that g does not contain an atomic formula $x = y$ with x, y distinct variables from $V(l)$, because then $g \Rightarrow x = y$ would be valid. Similarly, g does not contain an atomic formula $x \neq y$ with x, y distinct variables from $V(l)$, because then $g \Rightarrow x \neq y$ would be valid. Thus each atomic formula in g either contains variable in or variable out (or both). W.l.o.g. we assume that if an atomic formulas contain out, variable out occurs on the left, and otherwise variable in occurs on the left. Moreover, we assume w.l.o.g. that g does not contain atomic formulas $\mathsf{in} = x$ and $\mathsf{out} = x$, for some $x \in V(l)$ (in such a case we may replace $\mathsf{out} = x$ by $\mathsf{out} = \mathsf{in}$). Let g_{out} be the conjunction of all atomic formulas from g that contain out, and let g_{in} be the conjunction of all remaining atomic formulas from g. Then $g = g_{\mathsf{in}} \wedge g_{\mathsf{out}}$. Observe that g_{in} does not contain subformulas $\mathsf{in} = x$ and $\mathsf{in} = y$, for distinct $x, y \in V(l)$, because then $g \Rightarrow x = y$ would be valid. Similarly, g_{in} does not contain subformulas $\mathsf{in} = x$ and $\mathsf{in} \neq y$, for distinct $x, y \in V(l)$, because then $g \Rightarrow x \neq y$ would be valid. Finally, observe that g_{in} does not contain subformulas $\mathsf{in} = x$ and $\mathsf{in} \neq x$, for $x \in V(l)$, because this would contradict satisfiability of g. Condition (1) from the lemma now follows. Using a similar argument, we may prove condition (2). Condition (3) follows by contradiction. Suppose that $g_{\mathsf{in}} \equiv \mathsf{in} = x$, $\varrho(y) = \mathsf{in}$ and $\varrho(z) = x$, for $y, z \in V(l')$. Then $g \wedge (\bigwedge_{u \in V(l')} u' = \rho(u)) \Rightarrow y' = z'$ is valid, which is a contradiction. Conditions (4) follows via a similar argument. Condition (5) follows from our assumption that g does not contain atomic formulas $\mathsf{in} = x$ and $\mathsf{out} = x$, for $x \in V(l)$.

"$\Leftarrow$" Suppose $l \xrightarrow{i,g,\varrho,o} l'$ is a transition of $\mathcal{R}$. Let $\zeta : \mathcal{V} \to \mathbb{Z}$ be an injective function. Valuation ξ for $V(l) \cup \{\mathsf{in}, \mathsf{out}\}$ assigns distinct values to all variables, except when equality is required by g:

$$\xi(z) \quad = \quad \begin{cases} \zeta(x) & \text{if } z \equiv \mathsf{out} \wedge g_{\mathsf{out}} \equiv \mathsf{out} = x \\ \zeta(\mathsf{in}) & \text{if } g_{\mathsf{in}} \equiv \mathsf{in} = z \\ \zeta(z) & \text{otherwise} \end{cases}$$

It is routine to check that $\xi \models g$, which means that g is satisfiable. Now suppose that x and y are distinct variables in $V(l)$. Then $\xi \not\models g \Rightarrow x = y$, which implies that $g \Rightarrow x = y$ is not valid. Let $n \in \mathbb{Z}$ be a value with $n \neq \zeta(\mathsf{in})$ and $n \neq \zeta(\mathsf{out})$. Valuation $\overline{\xi}$ for $V(l) \cup \{\mathsf{in}, \mathsf{out}\}$ assigns the same value n to all variables, except when inequality is required by g:

$$\overline{\xi}(z) \quad = \quad \begin{cases} \zeta(\mathsf{in}) & \text{if } z \equiv \mathsf{in} \wedge g_{\mathsf{in}} \text{ contains an inequality} \\ \zeta(\mathsf{out}) & \text{if } z \equiv \mathsf{out} \wedge g_{\mathsf{out}} \text{ contains an inequality} \\ n & \text{otherwise} \end{cases}$$

It is again routine to check that $\overline{\xi} \models g$. Suppose that x and y are distinct variables in $V(l)$. Then $\overline{\xi} \not\models g \Rightarrow x \neq y$, which implies that $g \Rightarrow x = y$ is not valid.

In order to prove condition (2), suppose x, y are distinct variables from $V(l')$. Let, for $z \in V(l')$, $\xi'(z') = \xi(\varrho(z))$. Then $\xi \cup \xi' \models g \wedge (\bigwedge_{z \in V(l')} z' = \rho(z))$. By construction ξ is injective, except that $\xi(\mathsf{in}) = \xi(z)$ in case $g_{\mathsf{in}} \equiv \mathsf{in} = z$, and $\xi(\mathsf{out}) = \xi(z)$ in case $g_{\mathsf{out}} \equiv \mathsf{out} = z$. This observation, in combination with the fact that ϱ is injective and conditions (3) and (4) from the statement of the lemma, gives us that ξ' is injective. Therefore $\xi \cup \xi'$ does not satisfy $g \wedge (\bigwedge_{z \in V(l')} z' = \rho(z)) \Rightarrow x' = y'$, which implies that this formula is not valid. □

Lemma 4.5 implies that each outgoing transition from a location l of $\mathcal{R}$ may fire in each state (l, ξ) of $[\![\mathcal{R}]\!]$, provided we choose the right input and output values. This has as an important consequence that a location l is reachable in $\mathcal{R}$ iff a state (l, ξ) is reachable in $[\![\mathcal{R}]\!]$, for some ξ.

Corollary 4.2. *Let $\mathcal{R}$ be a right invariant RA with a transition $l \xrightarrow{i,g,\varrho,o} l'$. Let $\xi \in \mathsf{Val}(V(l))$. Then $[\![\mathcal{R}]\!]$ has a transition $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ that is supported by $l \xrightarrow{i,g,\varrho,o} l'$.*

Proof. We may assume that g is of the form $g_{\mathsf{in}} \wedge g_{\mathsf{out}}$ described in Lemma 4.5. If g_{in} of the form $\mathsf{in} = x$, for some $x \in V(l)$, then choose $d = \xi(x)$. Otherwise, let d be equal to some arbitrary fresh value outside the range of ξ. Similarly, pick a value for

e. Then with $\iota = \xi \cup \{(\mathsf{in}, d), (\mathsf{out}, e)\}$ we have $\iota \models g$ by construction, and thus (l, ξ) enables a transition that is supported by $l \xrightarrow{i,g,\varrho,o} l'$. □

Another restriction on register automata that plays an important role in our work is *unique-valuedness*. Intuitively, this means that registers are required to always store unique values.

Definition 4.7. *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma\rangle$ be a register automaton. Then $\mathcal{R}$ is* unique-valued *if, for each reachable state (l, ξ) of $[\![\mathcal{R}]\!]$, valuation ξ is injective, that is, two registers can never store identical values.*

Example 4.6. *The FIFO-set model of Figure 4.2 and the login model of Figure 4.3 are both unique-valued. The slot machine model of Figure 4.7 is not unique-valued, since in location l_2 registers v and w may contain the same value. Figure 4.8 presents a variation of the FIFO-set model that is right invariant but not unique-valued. This register automaton, which represents a FIFO-buffer of capacity 2, is not unique-valued since in location l_2 registers v and w may contain the same value.*

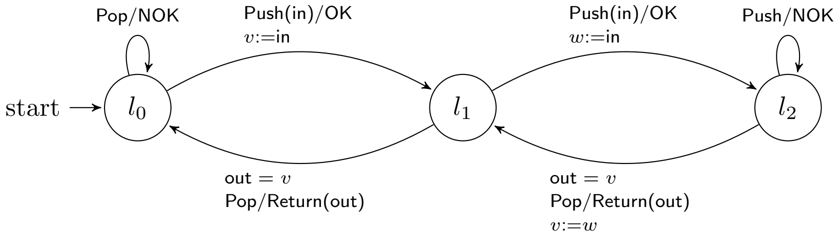


Figure 4.8: FIFO-buffer with a capacity of 2 modeled as a register automaton

Even though right invariance and unique-valuedness are strong restrictions it is possible to construct, for each register automaton, an equivalent register automaton that is both right invariant and unique valued. Figure 4.9, for example, shows a right invariant and unique-valued register automaton that is equivalent to the register automaton of Figure 4.7.

Theorem 4.1. *For each register automaton $\mathcal{R}$ there exists a right invariant and unique-valued register automaton $\overline{\mathcal{R}}$ such that $\mathcal{R}$ and $\overline{\mathcal{R}}$ are equivalent.*

Proof. See the extended version of this chapter available at [5]. □

Cassel et al [54] established that right invariant register automata can be exponentially more succinct than unique-valued register automata. The second main result of this section is that (arbitrary) register automata in turn can be exponentially more succinct than right invariant register automata.

Theorem 4.2. *There exists a sequence of register automata $\mathcal{R}_1$, $\mathcal{R}_2$,.. such that the*

[5]http://www.sws.cs.ru.nl/publications/papers/fvaan/TomteFresh/

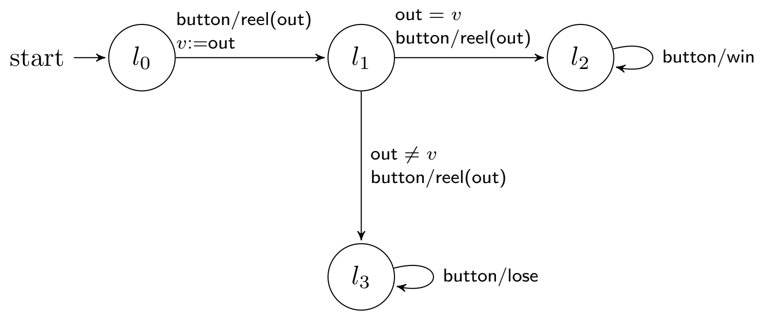


Figure 4.9: Slot machine modeled as a right invariant register automaton

number of locations of $\mathcal{R}_n$ is $O(n)$, but the minimal number of locations of a right invariant register automaton that is equivalent to $\mathcal{R}_n$ is $\Omega(2^n)$.

Proof. The idea is to let $\mathcal{R}_n$ encode a binary counter with n bits. The register automaton $\mathcal{R}_n$ has input symbols Init and Tick, output symbols OK and Overflow, $n+2$ locations $l_0, l_1, c_1, \ldots, c_n$, and $n+2$ registers $zero, one, x_n, \ldots, x_1$. Figure 4.10 shows the transitions of $\mathcal{R}_n$. We view Tick as the default input symbol, OK as the default output symbol, and do not display these default symbols in the diagram.

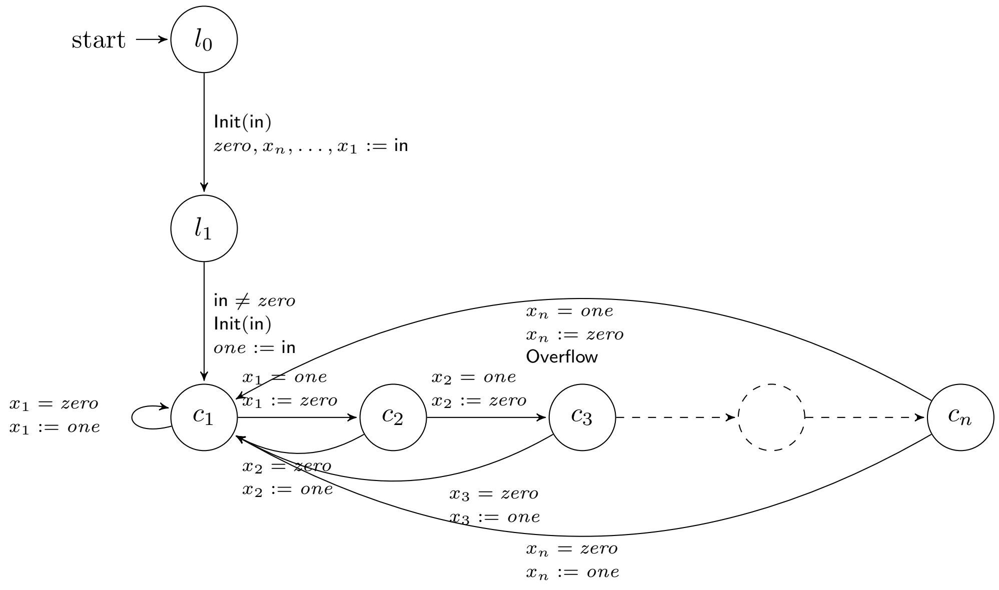


Figure 4.10: Encoding a binary counter as a register automaton

Mealy machine $[\![\mathcal{R}_n]\!]$ has runs in which repeatedly location c_1 is visited. The first time all the variables $x_n, \ldots, x_1$ equal $zero$, which encodes a binary counter with value 0, with x_1 representing the least significant bit. Then, for each subsequent visit to

c_1, the value of the counter is incremented by one. When the counter overflows all the bits become *zero* again and an output Overflow is generated. Since each cycle from c_1 to itself takes at least one transition, it takes at least 2^n transitions before an output Overflow occurs. By Theorem 4.1, we know that there exists a right invariant register automaton that is equivalent to $\mathcal{R}_n$. Let $\mathcal{R}'_n$ be such a right invariant register automaton with a minimal number of locations. Then $\mathcal{R}'_n$ has a transition with output symbol Overflow, starting from a location l that is reachable with a cycle free path of transitions in $\mathcal{R}'_n$. Due to Corollary 4.2, the number of transitions in this path is at least 2^n (otherwise $[\![\mathcal{R}'_n]\!]$ would be able to produce an Overflow prematurely). Hence $\mathcal{R}'_n$ contains at least 2^n locations. □

In this chapter, we will present an algorithm for learning input enabled, input deterministic, register automata. The models produced by our learning algorithm will be a right invariant register automaton. By the results of this section, such a right invariant register automaton will always exist, although it may be exponentially less succinct than the original register automaton of the teacher.

4.4 Model Learning

Algorithms for active learning of automata have originally been developed for inferring finite state acceptors for unknown regular languages [19]. Since then these algorithms have become popular with the testing and verification communities for inferring models of black box systems in an automated fashion. While the details change for concrete classes of systems, all of these algorithms follow basically the same pattern. They model the learning process as a game between a learner and a teacher. The learner has to infer an unknown automaton with the help of the teacher. The learner can ask three types of queries to the teacher:

Output Queries ask for the expected output for a concrete sequence of inputs. In practice, output queries can be realized as simple tests.

Reset Queries prompt the teacher to return to its initial state and are typically asked after each output query.

Equivalence Queries check whether a conjectured automaton produced by the learner is correct. In case the automaton is not correct, the teacher provides a counterexample, a trace exposing a difference between the conjecture and the expected behavior of the system to be learned. Equivalence queries can be approximated through (model-based) testing in black-box scenarios.

A learning algorithm will use these three kinds of queries and produce a sequence of automata converging towards the correct one in a finite number of steps. We refer the reader to [121, 198, 205] for introductions to active automata learning.

4.4.1 The Nerode Congruence

Most of the learning algorithms that have been proposed in the literature aim to construct an approximation of the Nerode congruence based on a finite number of queries. The famous Myhill-Nerode theorem [163] for Deterministic Finite Automata (DFA) provides a basis for describing (a) how prefixes traverse states (equivalence classes), and (b) how states can be distinguished (by suffixes). Below we present a straightforward reformulation of the Myhill-Nerode theorem for deterministic Mealy machines, adapted from [198].

Definition 4.8. *An* observation *over a set of inputs I and a set of outputs O is a finite alternating sequence $i_0 o_0 \cdots i_{n-1} o_{n-1}$ of inputs and outputs that is either empty, or begins with an input and ends with an output. Let S be a set of observations over I and O. Then S is*

- prefix closed *if $\beta\ i\ o \in S \implies \beta \in S$,*
- behavior deterministic *if $\beta\ i\ o \in S \wedge \beta\ i\ o' \in S \implies o = o'$, and*
- input complete *if $\beta \in S \wedge i \in I \implies \exists o \in O : \beta\ i\ o \in S$.*

Two observations $\beta, \beta' \in S$ are equivalent *for S, notation $\beta \equiv_S \beta'$, iff for all observations γ over I and O, $\beta\gamma \in S \Leftrightarrow \beta'\gamma \in S$. We write $[\beta]$ to denote the equivalence class of β with respect to $\equiv_S$.*

Theorem 4.3 (Myhill-Nerode)**.** *Let S be a set of observations over finite sets of inputs I and outputs O. Then S is the set of traces of some finite, deterministic Mealy machine $\mathcal{M}$ iff S is nonempty, prefix closed, behavior deterministic, input complete, and $\equiv_S$ has finitely many equivalence classes (finite index).*

Proof. "$\Rightarrow$". Let $\mathcal{M}$ be a finite, deterministic Mealy machine and let S be its set of traces. Then it is immediate from the definitions that S is a nonempty set of observations that is prefix closed and input complete. Since each trace of $\mathcal{M}$ leads to a unique state and $\mathcal{M}$ is deterministic, it follows that S is behavior deterministic. Since all observations that lead to the same state are obviously equivalent and since $\mathcal{M}$ is finite, equivalence relation $\equiv_S$ has finite index.

"$\Leftarrow$". Suppose S is nonempty, prefix closed, behavior deterministic, input complete, and $\equiv_S$ has finite index. We define the finite, deterministic Mealy machine $\mathcal{M} = \langle I, O, Q, q^0, \delta, \lambda \rangle$ as follows:

- Q is the set of classes of $\equiv_S$.
- q^0 is given by $[\epsilon]$.
- Let $\beta \in S$ and $i \in I$. Then, since S is both input complete and behavior deterministic, there exists a unique $o \in O$ such that $\beta\ i\ o \in S$. We define $\delta([\beta], i) = [\beta\ i\ o]$ and $\lambda([\beta], i) = o$.

It is straightforward to verify that $\mathcal{M}$ is a well-defined finite, deterministic Mealy machine whose set of traces equals S. □

The equivalence relation $\equiv_S$ induced by the set of traces S of a register automaton does not have a finite index. However, as observed by [35, 54, 55], by using the inherent symmetry of register automata we may define a slightly different equivalence relation $\equiv_S^{aut}$ that does have a finite index and that may serve as a basis for a Myhill-Nerode theorem for register automata. The equivalence relation $\equiv_S^{aut}$ on S is defined by

$$\beta \equiv_S^{aut} \beta' \quad \Leftrightarrow \quad \exists \text{ automorphism } h \; \forall \gamma : (\beta\gamma \in S \Leftrightarrow \beta' h(\gamma) \in S)$$

Proposition 4.1. *Let $\mathcal{R}$ be an input deterministic register automataton and let S be its set of traces. Then $\equiv_S^{aut}$ has a finite index.*

Proof. Since $\mathcal{R}$ is input deterministic, there exists for each trace of $\mathcal{R}$ a unique corresponding run. Let β and β' be traces of $\mathcal{R}$ and let (l, ξ) and (l', ξ') be the final states of the corresponding runs. Assume that $l = l'$ and $\mathsf{Part}(\xi) = \mathsf{Part}(\xi')$. (Here we write $\mathsf{Part}(f)$ for the partition induced by function f in which two elements from the domain of f are placed in the same block iff f maps them to the same value.) Then there exists an automorphism h from ξ to ξ'. By Lemma 4.3, α is a partial run starting in (l, ξ) iff $h(\alpha)$ is a partial run starting in (l', ξ'). Moreover, by Lemma 4.4, $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$. Hence, $\beta \equiv_S^{aut} \beta'$. Since $\mathcal{R}$ has a finite number of locations, since each location has a finite set of registers, and since there are only finitely many partitions of a finite set, this implies that $\equiv_S^{aut}$ has a finite index. □

Whereas [114,115] presents a learning algorithm for register automata that is based on a variant of the Myhill-Nerode theorem for $\equiv_S^{aut}$ (i.e., the "converse" of Proposition 4.1), the idea of our approach is to learn register automata by constructing an abstraction of the set of traces that has a finite index according to the original definition of $\equiv_S$.

4.4.2 Mappers

Below we recall relevant parts of the theory of mappers from [10]. In order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we may place a transducer in between the teacher and the learner, which translates concrete inputs to abstract inputs, concrete outputs to abstract outputs, and vice versa. This allows us to reduce the task of the learner to inferring a "small" Mealy machine with an abstract alphabet. As we will see, the determinizer and the abstractor of Figure 4.1 are examples of such transducers.

The behavior of a transducer is fully specified by a *mapper*, a deterministic Mealy machine in which concrete actions are inputs and abstract actions are outputs.

Definition 4.9 (Mapper). *A* mapper *is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where*

- *I and O are disjoint sets of* concrete input and output actions*,*
- *X and Y are disjoint sets of* abstract input and output actions*, and*

- $\lambda : R \times (I \cup O) \to (X \cup Y)$, *referred to as the* abstraction function*, respects inputs and outputs, that is, for all* $a \in I \cup O$ *and* $r \in R$, $a \in I \Leftrightarrow \lambda(r, a) \in X$.

A mapper $\mathcal{A}$ translates any sequence $\beta \in (I \cup O)^*$ of concrete actions into a corresponding sequence of abstract actions given by

$$\alpha_{\mathcal{A}}(\beta) \quad = \quad \lambda(r_0, \beta).$$

A mapper also allows us to abstract a Mealy machine with concrete actions in I and O into a Mealy machine with abstract actions in X and Y. 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 actions into abstract ones.

Definition 4.10 (Abstraction)**.** *Let* $\mathcal{M} = \langle I, O, Q, q_0, \to \rangle$ *be a Mealy machine and let* $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$ *be a mapper. Then* $\alpha_{\mathcal{A}}(\mathcal{M})$*, the* abstraction of $\mathcal{M}$ via $\mathcal{A}$*, is the Mealy machine* $\langle X, Y \cup \{\perp\}, Q \times R, (q_0, r_0), \to \rangle$*, where* $\perp \notin Y$ *is a fresh output action and* $\to$ *is given inductively by the rules*

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

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 $\lambda(r, i) = x$. In this case, there exist unique r', r'' and y such that $r \xrightarrow{i/x} r'$ and $r' \xrightarrow{o/y} r''$. An x-transition in state (q, r) then leads to state (q', r'') and produces output y. The second rule in the definition ensures 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 x there does not exist a corresponding concrete input i with $\lambda(r, i) = x$. In this case, an input x triggers the special "undefined" output action $\perp$ and leaves the state unchanged.

Lemma 4.6. *Let* $\mathcal{A}$ *be a mapper and let* $\mathcal{M}$ *be a Mealy machine with the same concrete input and output actions* I *and* O*. If* β *is a trace of* $\mathcal{M}$ *then* $\alpha_{\mathcal{A}}(\beta)$ *is a trace of* $\alpha_{\mathcal{A}}(\mathcal{M})$.

Proof. Straightforward, see also Lemma 4 of [10]. □

A mapper describes the behavior of a transducer component that we can place in between a Learner and a Teacher. Consider a mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$. The transducer component that is induced by $\mathcal{A}$ records the current state, which initially is set to r_0, and behaves as follows:

- Whenever the transducer is in a state r and receives an abstract input $x \in X$ from the learner, it nondeterministically picks a concrete input $i \in I$ such that $\lambda(r, i) = x$, forwards i to the teacher, and jumps to state $\delta(r, i)$. If there exists no such input i, then the component returns output $\perp$ to the learner.

- Whenever the transducer is in a state r and receives a concrete answer o from the teacher, it forwards $\lambda(r, o)$ to the learner and jumps to state $\delta(r, o)$.
- Whenever the transducer receives a reset query from the learner, it changes its current state to r_0, and forwards a reset query to the teacher.

From the perspective of a learner, a teacher for $\mathcal{M}$ and a transducer for $\mathcal{A}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. (We refer to [10] for a formalization of this claim.) In [10], also a *concretization* operator $\gamma_{\mathcal{A}}$ is defined. Let $\mathcal{H}$ be a Mealy machine with "abstract" actions in X and Y. The concretization operator $\gamma_{\mathcal{A}}$ is the adjoint of the abstraction operator: it turns $\mathcal{H}$ into a concrete Mealy machine $\gamma_{\mathcal{A}}(\mathcal{H})$ with actions in I and O. As shown in [10], $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.

4.5 The Determinizer

The login example of Figure 4.3 shows that input deterministic register automata may exhibit nondeterministic behavior: in each run the automaton may generate different output values (passwords). This is a useful feature since it allows us to model the actual behavior of real-world systems, but it is also problematic since learning tools such as LearnLib can only handle deterministic systems. In this section, we show how this type of nondeterminism can be eliminated by exploiting symmetries that are present in register automata.

As a first step, we show that each trace is equivalent to a 'neat' trace in which fresh values are selected according to some fixed rules. The concept of 'neat' traces is similar to the encoding with 'representative' traces that was used in [55] in a setting without fresh values.

Definition 4.11 (Fresh and neat). *Consider a trace β of register automaton $\mathcal{R}$:*

$$\beta \quad = \quad i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1}) \tag{4.2}$$

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur before output o_j, that is, $S_0 = \emptyset$, $T_j = S_j \cup \{d_j\}$ and $S_{j+1} = T_j \cup \{e_j\}$. An input value d_j is fresh *if it has not occurred before in the trace, that is, $d_j \notin S_j$. Similarly, an output value e_j is* fresh *if it has not occurred before, that is, $e_j \notin T_j$. We say that β has* neat inputs *if each fresh input value d_j is equal to the largest preceding value (including 0) plus one, that is, $d_j \in S_j \cup \{\max(S_j \cup \{0\}) + 1\}$. Similarly, β has* neat outputs *if each fresh output value is equal to the smallest preceding value (including 0) minus one, that is, for all j, $e_j \in T_j \cup \{\min(T_j \cup \{0\}) - 1\}$. A trace is* neat *if it has neat inputs and neat outputs, and a run is* neat *if its trace is neat.*

Observe that in a neat trace the n-th fresh input value is n, and the n-th fresh output value is $-n$.

Example 4.7. *Trace $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ is not neat, for instance because the first fresh output value 3 is not equal to -1. Also, the second input value 7 is fresh* *but different from* 4*, the largest preceding value plus* 1*. An example of a neat trace is* $i(1)\ o(-1)\ i(2)\ o(2)\ i(-1)\ o(-2)$*.*

The next proposition implies that in order to learn the behavior of a register automaton it suffices to study its neat traces, since any other trace is equivalent to a neat trace. In order to prove this result, we need the following technical definition, which extends any finite one-to-one relation to an automorphism.

Definition 4.12. *For each finite set* $S \subseteq \mathbb{Z}$*, let* $\mathsf{EnumCompl}(S)$ *be a function that enumerates the elements in the complement of* S*, that is,* $\mathsf{EnumCompl}(S) : \mathbb{N} \to (\mathbb{Z} \setminus S)$ *is a bijection. Then, for any finite one-to-one relation* $r \subseteq \mathbb{Z}\times\mathbb{Z}$*,* $\hat{r}$ *is the automorphism given by:*

$$\hat{r} \quad = \quad r \cup \{(\mathsf{EnumCompl}(\mathsf{dom}(r))(k), \mathsf{EnumCompl}(\mathsf{ran}(r))(k)) \mid k \in \mathbb{N}\}.$$

Here $\mathsf{dom}(r)$ *denotes the domain of* r *and* $\mathsf{ran}(r)$ *denotes the range of* r*.*

Proposition 4.2. *For every trace* β *there exists a zero respecting automorphism* h *such that* $h(\beta)$ *is neat.*

Proof. Let β, S_j and T_j $(j = 0, \ldots, n-1)$ be as in Definition 4.11. Inductively, we define relations $s_j, t_j \subseteq \mathbb{Z} \times \mathbb{Z}$ (for $j = 0, \ldots, n-1$) as follows

$$\begin{array}{rcl} s_0 & = & \emptyset \\ t_j & = & \begin{cases} s_j \cup \{(d_j, \max(\mathsf{ran}(s_j) \cup \{0\}) + 1)\} & \text{if } d_j \text{ is fresh} \\ s_j & \text{otherwise} \end{cases} \\ s_{j+1} & = & \begin{cases} t_j \cup \{(e_j, \min(\mathsf{ran}(t_j) \cup \{0\}) - 1)\} & \text{if } e_j \text{ is fresh} \\ t_j & \text{otherwise} \end{cases} \end{array}$$

By induction, we can prove the following assertions, for all j: (1) $\mathsf{dom}(s_j) = S_j$ and $\mathsf{dom}(t_j) = T_j$, (2) s_j and t_j are injective. By construction, $t_{n-1}(\beta)$ is neat. Then $h = \hat{t}_{n-1}$ is an automorphism such that $h(\beta)$ is neat. □

Example 4.8. *Consider the trace* $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ *from Example 4.7. This non neat trace can be mapped to the neat trace* $i(1)\ o(-1)\ i(2)\ o(2)\ i(-1)\ o(-2)$ *by the automorphism* h *that acts as the identity function except that it permutes some values:* $h(3) = -1$*,* $h(-1) = 7$*,* $h(7) = 2$*,* $h(2) = -2$*, and* $h(-2) = 3$*.*

Corollary 4.3. *For every run* α *of* $\mathcal{R}$ *there exists an automorphism* h *such that* $h(\alpha)$ *is neat.*

Proof. Let α be a run of $\mathcal{R}$. Then $\beta = \mathsf{trace}(\alpha)$ is a trace of $\mathcal{R}$. Therefore, by Proposition 4.2, there exists an automorphism h such that $h(\beta)$ is neat. By Lemma 4.3, $h(\alpha)$ is a run of $\mathcal{R}$ and by Lemma 4.4, $\mathsf{trace}(h(\alpha)) = h(\beta)$. Since $h(\beta)$ is neat and a run is neat if its trace is neat, $h(\alpha)$ is neat as well. □

Whereas the learner may choose to only provide neat inputs, we usually have no control over the outputs generated by the SUL, so in general these will not be neat.

In order to handle this, we place a component, called the *determinizer*, in between the SUL and the learner. The determinizer renames the outputs generated by the SUL and makes them neat. The behavior of the determinizer is specified by the mapper $\mathcal{D}$ defined below. As part of its state $\mathcal{D}$ maintains a finite one-to-one relation r describing the current renamings, which grows dynamically during an execution (similar to the functions s_j and t_j in the proof of Proposition 4.2). We write $\hat{r}$ for an automorphism that extends r (we may construct $\hat{r}$ using the construction described in the proof of Proposition 4.2). Whenever the SUL generates an output n that does not occur in $\mathsf{dom}(r)$, this output is mapped to a value m one less than the minimal value in $\mathsf{ran}(R)$, and the pair (n, m) is added to r. Whenever the learner generates an input m, the mapper concretizes this value to $n = \hat{r}^{-1}(m)$ and forwards n to the SUL. If n does not occur in $\mathsf{dom}(r)$, then r is extended with the pair (n, m).

Definition 4.13 (Determinizer). *Let I and O be finite, disjoint sets of input and output symbols. The* determinizer *for I and O is the mapper $\mathcal{D} = \langle (I \times \mathbb{Z}) \cup (O \times \mathbb{Z}), (I \times \mathbb{Z}) \cup (O \times \mathbb{Z}), R, r_0, \delta, \lambda \rangle$ where*

- $R = \{r \subseteq \mathbb{Z} \times \mathbb{Z} \mid r \textit{ finite and one-to-one}\}$,
- $r_0 = \emptyset$,
- *for all $r \in R$, $i \in I$, $o \in O$ and $n \in \mathbb{Z}$,*

$$\begin{aligned}
\lambda(r, i(n)) &= i(\hat{r}(n)) \\
\lambda(r, o(n)) &= \begin{cases} o(r(n)) & \text{if } n \in \mathsf{dom}(r) \\ o(\min(\mathsf{ran}(r) \cup \{0\}) - 1) & \text{otherwise} \end{cases} \\
\delta(r, i(n)) &= \begin{cases} r & \text{if } n \in \mathsf{dom}(r) \\ r \cup \{(n, \hat{r}(n))\} & \text{otherwise} \end{cases} \\
\delta(r, o(n)) &= \begin{cases} r & \text{if } n \in \mathsf{dom}(r) \\ r \cup \{(n, \min(\mathsf{ran}(r) \cup \{0\}) - 1)\} & \text{otherwise} \end{cases}
\end{aligned}$$

Proposition 4.3. *Let $\mathcal{R}$ be a register automaton with inputs I and outputs O, let $\mathcal{D}$ be the determinizer for I and O, and let β be a trace of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$. Then β has neat outputs and is equivalent to a trace of $\mathcal{R}$.*

Proof. Let α be a run of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$ with trace β. We claim that α does not contain any transitions with output $\perp$, that is, transitions generated by the second rule in Definition 4.9. This is because, for any state r of mapper $\mathcal{D}$ and any 'abstract' input $i(d)$, there exists a 'concrete' input $i(d')$ such that $\lambda(r, i(d')) = i(d)$. In fact, since $\hat{r}$ is an automorphism, we can just take $d' = \hat{r}^{-1}(d)$. Hence run α takes the form

$$\alpha = ((l_0, \xi_0), r_0)\ i_0(d_0)\ o_0(e_0)\ ((l_1, \xi_1), r_1)\ i_1(d_1)\ o_1(e_1)\ ((l_2, \xi_2), r_2) \cdots$$
$$\cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ ((l_n, \xi_n), r_n).$$

Since the transitions in run α have been derived by repeated application of the first rule in Definition 4.9, there exist d'_j, e'_j and r'_j such that $[\![\mathcal{R}]\!]$ has a run α' of the form

$$\alpha' = (l_0, \xi_0)\ i_0(d'_0)\ o_0(e'_0)\ (l_1, \xi_1)\ i_1(d'_1)\ o_1(e'_1)\ (l_2, \xi_2) \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ o_{n-1}(e'_{n-1})\ (l_n, \xi_n),$$

and $\mathcal{D}$ has a run

$$r_0\ i_0(d'_0)\ i_0(d_0)\ r'_0\ o_0(e'_0)\ o_0(e_0)\ r_1\ i_1(d'_1)\ i_1(d_1)\ r'_1\ o_1(e'_1)\ o_1(e_1)\ r_2 \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ i_{n-1}(d_{n-1})\ r'_{n-1}\ o_{n-1}(e'_{n-1})\ o_{n-1}(e_{n-1})\ r_n.$$

From Definition 4.13 we may infer that, for all $j < n$, $(d'_j, d_j) \in r'_j$, $(e'_j, e_j) \in r_{j+1}$, $r_j \subseteq r'_j$ and $r'_j \subseteq r_{j+1}$. Now let $h = \hat{r}_n$. Then h is an automorphism satisfying, for all $j < n$, $h(d'_j) = d_j$ and $h(e'_j) = e_j$. Let β' be the trace of α'. Then $h(\beta') = \beta$ and thus traces β and β' are equivalent.

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur in β before output o_j. Then it follows by induction that $S_j = \mathsf{ran}(r_j)$ and $T_j = \mathsf{ran}(r'_j)$. According to Definition 4.11, β has neat outputs if $e_j \in T_j \cup \{\min(T_j \cup \{0\}) - 1\}$, that is, if $e_j \in \mathsf{ran}(r'_j) \cup \{\min(\mathsf{ran}(r'_j \cup \{0\})) - 1\}$. But this is implied by Definition 4.13. □

Proposition 4.4. *Any trace of $\mathcal{R}$ with neat outputs is also a trace of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$.*

Proof. Let α be a run of $[\![\mathcal{R}]\!]$ with trace β. Then run α takes the form

$$\alpha = (l_0, \xi_0)\ i_0(d_0)\ o_0(e_0)\ (l_1, \xi_1)\ i_1(d_1)\ o_1(e_1)\ (l_2, \xi_2) \cdots$$

$$\cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ (l_n, \xi_n).$$

$\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$ has a corresponding run α' of the form

$$\alpha' = ((l_0, \xi_0), r_0)\ i_0(d'_0)\ o_0(e'_0)\ ((l_1, \xi_1), r_1)\ i_1(d'_1)\ o_1(e'_1)\ ((l_2, \xi_2), r_2) \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ o_{n-1}(e'_{n-1})\ ((l_n, \xi_n), r_n)$$

and $\mathcal{D}$ has a run

$$r_0\ i_0(d_0)\ i_0(d'_0)\ r'_0\ o_0(e_0)\ o_0(e'_0)\ r_1\ i_1(d_1)\ i_1(d'_1)\ r'_1\ o_1(e_1)\ o_1(e'_1)\ r_2 \cdots$$

$$\cdots\ i_{n-1}(d_{n-1})\ i_{n-1}(d'_{n-1})\ r'_{n-1}\ o_{n-1}(e_{n-1})\ o_{n-1}(e'_{n-1})\ r_n.$$

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur in β before output o_j. Then it follows by induction that $S_j = \mathsf{dom}(r_j)$ and $T_j = \mathsf{dom}(r'_j)$. Since β has neat outputs, $e_j \in \mathsf{dom}(r'_j) \cup \{\min(\mathsf{dom}(r'_j) \cup \{0\}) - 1\}$. Let Id denote the identity function on $\mathbb{Z}$, that is, $Id = \{(n, n) \mid n \in \mathbb{Z}\}$. Observe that for any finite one-to-one relation $r \subseteq Id$, $\hat{r} = Id$. By induction on j, we may now prove that $r_j, r'_j \subseteq Id$. It follows that $d_j = d'_j$ and $e_j = e'_j$, for all j. Thus β is a trace of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$, as required. □

Corollary 4.4. *$\mathcal{R}$ and $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$ have equivalent traces.*

Proof. Immediate from Propositions 4.2, 4.3 and 4.4. □

Example 4.9. *The determinizer does not remove all sources of nondeterminism. The login model of Figure 4.3, for instance, is not behavior deterministic, even when we only consider neat traces, because of neat traces* Register(1) OK(1) *and* Register(1) OK(−1)*. This nondeterminism may be considered 'harmless' since the parameter value of the* OK*-output is not stored and the behavior after the different outputs is the same. The slot machine model of Figure 4.7, however, has real nondeterminism in the sense that traces* button(1) reel(−1) button(1) reel(−2) *and* button(1) reel(−1) button(1) reel(−1) *lead to states with distinct output symbols in the outgoing transitions.*

The slot machine of Example 4.9 nondeterministically selects an output which 'accidentally' may be equal to a previous value. We call this a *collision*.

Definition 4.14. *Let β be a trace of register automaton $\mathcal{R}$. Then β* ends with a collision *if (a) the last output value is not fresh, and (b) the sequence obtained by replacing this value by some other value is also a trace of $\mathcal{R}$. We say that β* has a collision *if it has a prefix that ends with a collision.*

Example 4.10. *Trace* button*(3)* reel*(137)* button*(8)* reel*(137) of the slot machine model of Figure 4.7 has a collision, because the last output value* 137 *is not fresh, and if we replace it by* 138 *the result is again a trace.*

In many protocols, fresh output values are selected from a finite but large domain. TCP sequence and acknowledgement numbers, for instance, comprise 32 bits. The traces generated during learning are usually not so long and typically contain only a few fresh outputs. As a result, chances that collisions occur during learning are typically negligible. For these reasons, we have decided to consider only observations without collisions. Under the assumption that the SUL will not repeatedly pick the same fresh value, we can detect whether an observation contains a collision by simply repeating experiments a few times: if, after the renaming performed by the determinizer, we still observe nondeterminism then a collision has occurred. By ignoring traces with collisions, it may occur that the models that we learn incorrectly describe the behavior of the SUL in the case of collisions. We will, for instance, miss the win-transition in the slot machine of Figure 4.7. But if collisions are rare then it is extremely difficult to learn those types of behavior anyway. In applications with many collisions (for instance when fresh outputs are selected randomly from a small domain) one should not use the approach in this chapter, but rather an algorithm for learning nondeterministic automata such as the one presented in [211].

Our approach for learning register automata with fresh outputs relies on the following proposition.

Proposition 4.5. *The set S of collision free neat traces of an input deterministic register automaton $\mathcal{R}$ is behavior deterministic.*

Proof. Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be an input deterministic register automaton and let S be the set of collision free neat traces of $\mathcal{R}$. Suppose that $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$

are traces in S. Our task is to prove that $o(e) = o'(e')$. Since $\mathcal{R}$ is input deterministic, there is a unique run α of $[\![\mathcal{R}]\!]$ with trace β. Let (l, ξ) be the last state of this run. Since $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$ are traces of $\mathcal{R}$, $[\![R]\!]$ has transitions $(l, \xi) \xrightarrow{i(d)/o(e)} (l_1, \xi_1)$ and $(l, \xi) \xrightarrow{i(d)/o'(e')} (l'_1, \xi'_1)$. Since $\mathcal{R}$ is input deterministic, there is a unique transition that supports both transitions of $[\![\mathcal{R}]\!]$ and thus $o = o'$. We consider two cases. If both values e and e' are fresh then, since traces $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$ are neat, e and e' are both equal to the smallest preceding value minus one and thus $e = e'$. Now assume that at least one value, say e, is not fresh. Then, since $\beta\ i(d)\ o(e)$ is collision free, no sequence obtained from $\beta\ i(d)\ o(e)$ by replacing e by some other value can be a trace of $\mathcal{R}$. Thus $e = e'$ also in this case. We conclude $o(e) = o'(e')$, as required. $\square$

Our learning approach works for those register automata in which, when a fresh output is generated, it does not matter for the future behavior whether or not this fresh output equals some value that occurred previously. This is typically the case for real-world systems such as servers that generate fresh identifiers, passwords or sequence numbers. The slot machine of Figure 4.7 and Figure 4.9 is an example of a system that we cannot learn.

Proposition 4.6. *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two input deterministic right invariant register automata in which* out *does not occur negatively in guards. Then $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent iff they have the same sets of collision free traces.*

4.6 The Lookahead Oracle

The main task of the lookahead oracle is to compute for each trace of the SUL a set of values that are memorable after occurrence of this trace. Intuitively, a value d is memorable if it has an impact on the future behavior of the SUL: either d occurs in a future output, or a future output depends on the equality of d and a future input. The notion of a memorable value is fundamental for register automata and was previously studied e.g. in [31].

Definition 4.15. *Let $\mathcal{R}$ be a register automaton, let β be a trace of $\mathcal{R}$, and let $d \in \mathbb{Z}$ be a parameter value that occurs in β. Then d is* memorable *after β iff there exists a* witness *for d, that is, a sequence β' such that $\beta\ \beta'$ is a trace of $\mathcal{R}$ and if we replace each occurrence of d in β' by a fresh value f then the resulting sequence $\beta\ (\beta'[f/d])$ is not a trace of $\mathcal{R}$ anymore.*

Example 4.11. *In the example of Figure 4.2, the set of memorable values after trace $\beta = \mathsf{Push}(1)\ \mathsf{OK}\ \mathsf{Push}(2)\ \mathsf{OK}\ \mathsf{Push}(3)\ \mathsf{NOK}$ is $\{1, 2\}$. Values 1 and 2 are memorable, because of the witness $\beta' = \mathsf{Pop}\ \mathsf{Return}(1)\ \mathsf{Pop}\ \mathsf{Return}(2)$. Sequence $\beta\ \beta'$ is a trace of the model, but if we rename either the 1 or the 2 in β' into a fresh value, then this is no longer the case. In the example of Figure 4.3, value 2207 is memorable after $\mathsf{Register}\ \mathsf{OK}(2207)$ because $\mathsf{Register}\ \mathsf{OK}(2207)\ \mathsf{Login}(2207)\ \mathsf{OK}$ is a trace of the* *automaton, but* $\mathsf{Register}$ $\mathsf{OK}(2207)$ $\mathsf{Login}(1)$ OK *is not.*

The next theorem gives a state based characterization of memorable values: a value d is memorable after a run of a deterministic register automaton iff the final state of that run is inequivalent to the state obtained by replacing all occurrences of f by a fresh value. Thus we can also say that a value d is memorable in a state of a register automaton.

Theorem 4.4. *Let $\mathcal{R}$ be a deterministic register automaton, let α be a run of $\mathcal{M}$ with $\mathsf{trace}(\alpha) = \beta$, let (l, ξ) be the last state of α, let $d \in \mathbb{Z}$, and let $f \neq d$ be a fresh value that does not occur in α. Let $\mathsf{swap}_{d,f}$ be the automorphism that maps d to f, f to d, and acts as identity for all other values. Then d is memorable after β iff $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$.*

Proof. Suppose d is memorable after β. Then there exists a witness for d, that is, a sequence β' such that $\beta\ \beta'$ is a trace of $\mathcal{R}$ and $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$. Since $\mathcal{R}$ is deterministic, α is the unique run of $\mathcal{M}$ with $\mathsf{trace}(\alpha) = \beta$. Therefore, since $\beta\ \beta'$ is a trace of $\mathcal{R}$, there exists a partial run α' that starts in (l, ξ) such that $\mathsf{trace}(\alpha') = \beta'$. Moreover, since $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$, $\mathsf{swap}_{d,f}(\beta')$ is not a trace of (l, ξ). By Lemma 4.3, $\mathsf{swap}_{d,f}(\alpha')$ is a partial run of $\mathcal{R}$ that starts in $(l, \mathsf{swap}_{d,f}(\xi))$. By Lemma 4.4, $\mathsf{trace}(\mathsf{swap}_{d,f}(\alpha')) = \mathsf{swap}_{d,f}(\beta')$. Thus $\mathsf{swap}_{d,f}(\beta')$ is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$, which in turn implies $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$.

For the other direction, suppose $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$. Then there exists a sequence β' that is a trace of (l, ξ) but not of $(l, \mathsf{swap}_{d,f}(\xi))$. We claim that β' is a witness for d. Clearly, $\beta\ \beta'$ is a trace of $\mathcal{R}$. Now suppose $\beta\ \mathsf{swap}_{d,f}(\beta')$ is a trace of $\mathcal{R}$. Then, since $\mathcal{R}$ is deterministic, $\mathsf{swap}_{d,f}(\beta')$ is a trace of (l, ξ). By Lemmas 4.3 and 4.4, $\mathsf{swap}_{d,f}(\mathsf{swap}_{d,f}(\beta'))$ is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$. Therefore, since $\mathsf{swap}_{d,f}$ is its own inverse, β' is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$, and we have derived a contradiction. Thus our assumption was wrong and $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$. □

The above theorem reduces the problem of deciding whether a value is memorable to the problem of deciding equivalence of two states in a register automaton. It is not hard to see that conversely the problem of deciding equivalence of states can be reduced to the problem of deciding whether a value is memorable. The problem of finding a witness for a memorable value is thus equivalent to the problem of finding a distinguishing trace between two states.

Consider the architecture of Figure 4.1. Whenever the Lookahead Oracle receives an input from the Abstractor, this is just forwarded to the Determinizer. However, when the Lookahead Oracle receives a concrete output o from the Determinizer, then it forwards o to the Abstractor, together with a list of the memorable values after the occurrence of o. The ordering of the memorable values in the list determines in which registers the values will be stored by the Abstractor. Different orderings are possible, and the choice of the ordering affects the size of the register automaton that we will learn (similar to the way in which the variable ordering affects the size of a Binary Decision Diagram [50]). Within the Tomte tool we have experimented with different orderings. A simple way to order the values, for instance, is to sort them in ascending order. An ordering that works rather well in practice, and on which we elaborate below, is the order in which the values occur in the run.

Let $\mathcal{R}$ be the input deterministic register automaton that we want to learn, and let β be a trace of $\mathcal{R}$. Then, since $\mathcal{R}$ is input deterministic, it has a unique run

$$\alpha = (l_0, \xi_0)\ i_0(d_0)\ o_0(e_0)\ (l_1, \xi_1)\ i_1(d_1)\ o_1(e_1)\ (l_2, \xi_2) \cdots$$
$$\cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ (l_n, \xi_n).$$

such that $\mathsf{trace}(\alpha) = \beta$. For $j \leq n$, we define $r_j \in \mathbb{Z}^*$ inductively as follows: $r_0 = \epsilon$ and, for $j > 0$, r_j is obtained from r_{j-1} by first appending d_{j-1} and/or e_{j-1} in case these values do not occur in the sequence yet, and then erasing all values that are not memorable in state (l_j, ξ_j). Then the task of the Lookahead Oracle is to annotate each output action of β with the list of memorable values of the state reached by doing this output:

$$\mathsf{Oracle}_{\mathcal{R}}(\beta) \quad = \quad i_0(d_0)\ o_0(e_0 r_1)\ i_1(d_1)\ o_1(e_1 r_2) \cdots i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1} r_n).$$

In order to accomplish its task, the Lookahead Oracle stores all the traces of the SUL observed during learning in an *observation tree.*

Definition 4.16. *An* observation tree *is a pair $(\mathcal{N}, MemV)$, where $\mathcal{N}$ is a finite, nonempty, prefix-closed set of collision free, neat traces, and function $MemV : \mathcal{N} \to \mathbb{Z}^*$ associates to each trace a finite sequence of distinct values which are memorable after running this trace.*

In practice, observation trees are also useful as a cache for repeated queries on the SUL. Figure 4.11 shows two observation trees for our FIFO-set example. For each trace β_j a list of memorable values is given.

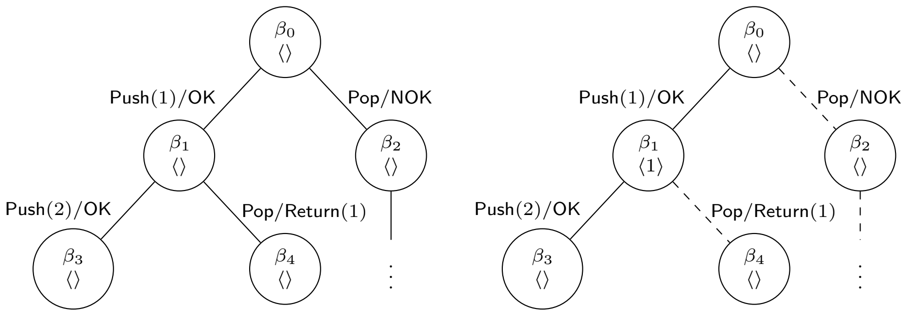


Figure 4.11: Observation trees for FIFO-set without and with Pop lookahead trace

Whenever a new trace β is added to the tree, the oracle computes a list of memorable values for it. For this purpose, the oracle maintains a list $L = \langle \sigma_1, \ldots, \sigma_k \rangle$ of *lookahead* *traces.* These lookahead traces are run in sequence after β to explore the future of β and to discover its memorable values.

Definition 4.17. *A* lookahead trace *is a sequence of symbolic input actions of the form $i(v)$ with $i \in I$ and $v \in \{p_1, p_2, \ldots\} \cup \{n_1, n_2, \ldots\} \cup \{f_1, f_2, \ldots\}$.*

Intuitively, a lookahead trace is a symbolic trace, where each parameter refers to either a previous value (p_j), or to a fresh input value (n_j), or to a fresh output value (f_j). Within lookahead traces, parameter p_1 plays a special role as the parameter that is replaced by a fresh value. Let σ be a lookahead trace in which parameters P refer to previous values, and let ζ be a valuation for P. Then σ can be converted into a concrete trace on the fly, by replacing each variable $p_j \in P$ by $\zeta(p_j)$, picking a fresh value for each variable n_j whenever needed, and assigning to f_j the j-th fresh output value. If trace γ is a possible outcome of converting lookahead trace σ, starting from a state (l, ξ) with valuation ζ, then we say that γ is a *concretization* of σ.

The following lemma implies that a finite number of lookahead traces will suffice to discover all memorable values of all states in an observation tree. The idea is that if a concretization of a lookahead trace is a witness that a value is memorable in some state, the same lookahead trace can also be used to discover that a corresponding value is memorable in any symmetric state.

Lemma 4.7. *Let $\mathcal{R}$ be a register automaton and let (l, ξ) be a state of $[\![\mathcal{R}]\!]$. Let σ be a lookahead trace in which parameters $P = \{p_1, \ldots, p_l\}$ refer to previous values, and let ζ be a valuation that assigns to each parameter in P a distinct memorable value of (l, ξ). Suppose γ is a concretization of σ starting from (l, ξ) with valuation ζ, and suppose γ is also a witness showing that $\zeta(p_1)$ is memorable in state (l, ξ). Let h be an automorphism and suppose γ' is a concretization of σ starting from state $h(l, \xi)$ with valuation $h \circ \zeta$. Then γ' is a witness showing that $h(\zeta(p_1))$ is memorable in state $h(l, \xi)$.*

If M is an overapproximation of the set of memorable values after some state (l', ξ') then, by concretizing lookahead trace σ for each injective valuation in $P \to M$, Lemma 4.7 guarantees that we will find a witness in case there exists an automorphism h from (l, ξ) to (l', ξ').

Instances of all lookahead traces are run in each new node to compute memorable values. At any point in time, the set of values that occur in $MemV(\beta)$ is a subset of the full set of memorable values of node β. Whenever a memorable value has been added to the observation tree, we require the tree to be *lookahead complete.* This means every memorable value has to have an origin, that is, it has to stem from either the memorable values of the parent node or the values in the preceding transition:

$$\beta' = \beta\ i(d)\ o(e) \quad \Rightarrow \quad \mathsf{values}(MemV(\beta')) \subseteq \mathsf{values}(MemV(\beta)) \cup \{d, e\},$$

where function values returns the set of elements that occur in a list. We employ a similar restriction on any non-fresh output parameters contained in the transition leading up to a node. These too have to originate from either the memorable values of the parent, or the input parameter in the transition. Herein we differentiate from the algorithm in [2] which only enforced this restriction on memorable values at the expense of running additional lookahead traces.

The observation tree at the left of Figure 4.11 is not lookahead complete since output value 1 of action $\mathsf{Return}(1)$ is neither part of the memorable values of the node β_1 nor is it an input in Pop. Whenever we detect such an incompleteness, we add a new lookahead trace (in this case Pop) and restart the entire learning process with the updated set of lookahead traces to retrieve a lookahead complete observation tree. The observation tree at the right is constructed after adding the lookahead trace Pop. This trace is executed for every node constructed, as highlighted by the dashed edges. The output values it generates are then tested if they are memorable and if so, stored in the $MemV$ set of the node. When constructing node β_1, the lookahead trace Pop gathers the output 1. This output is verified to be memorable and then stored in $MemV(\beta_1)$. We refer to [2] for more details about algorithms for the lookahead oracle.

4.7 The Abstractor

The task of the abstractor is to rename the large set of concrete values of the SUL to a small set of symbolic values that can be handled by the learner.

Let $w_0, w_1, \ldots$ be an enumeration of the set $\mathcal{V} \setminus \{\mathsf{in}, \mathsf{out}\}$. If the SUL can be described by a register automaton in which each location has at most n variables, then the abstract values used by the abstractor will be contained in $\{w_0, \ldots, w_{n-1}, \bot\}$. We define a family of mappers $\mathcal{A}_F$, which are parametrized by a function F that assigns to each input symbol a finite set of variables from $\mathcal{V} \setminus \{\mathsf{in}, \mathsf{out}\}$. Intuitively, $w \in F(i)$ indicates that it is relevant whether the parameter of input symbol i is equal to w or not. The initial mapper is parametrized by function $F_\emptyset$ that assigns to each input symbol the empty set. Using counterexample-guided abstraction refinement, the sets $F(i)$ are subsequently extended.

The states of $\mathcal{A}_F$ are injective sequences of values (that is, sequences in which each value occurs at most once), with the initial state being equal to the empty sequence. A sequence $r = d_0 \ldots d_{n-1} \in \mathbb{Z}^*$ represents the valuation ξ_r for $\{w_0, \ldots, w_{n-1}\}$ given by $\xi_r(w_j) = d_j$, for all j. Note that r is injective iff ξ_r is injective. The abstraction function of mapper $\mathcal{A}_F$ leaves the input and output symbols unchanged, but modifies the parameter values. The actual value of an input parameter is replaced by the variable in $F(i)$ that has the same value, or by $\bot$ in case there is no such variable. Thus the abstract domain of the parameter of i is the finite set $F(i) \cup \{\bot\}$. Likewise, the actual value of an output parameter is not preserved, but only the name of the variable that has the same value, or $\bot$ if there is no such variable. The (injective) sequence r' of memorable values that has been added as an annotation by the lookahead oracle describes the new state of the mapper after an output action. The abstraction function replaces r' by an update function ϱ that specifies how r' can be computed from the old state r and the input and output values that have been received. Upon receipt of a concrete output $o(e\ r')$ from the lookahead oracle, the abstraction function replaces e by a variable that is equal to e, or to $\perp$ if no such variable exists.

Definition 4.18. *We define* $\mathcal{A}_F = \langle I' \cup O', X \cup Y, R, r_0, \delta, \lambda \rangle$ *where*

- $I' = I \times \mathbb{Z}$,
- $O' = \{o(d\ r) \mid o \in O \wedge d \in \mathbb{Z} \wedge r \in \mathbb{Z}^* \text{ injective}\}$,
- $X = \{i(a) \mid i \in I \wedge a \in F(i) \cup \{\perp\}\}$,
- $Y = \{o(a, \varrho) \mid o \in O \wedge a \in \mathcal{V} \cup \{\perp\} \wedge \varrho \in \mathcal{V} \nrightarrow \mathcal{V} \text{ injective with finite domain}\}$,
- $R = \{r \in \mathbb{Z}^* \mid r \text{ injective}\}$,
- $r_0 = \epsilon$,
- $\delta(r, i(d)) = d\ r$,
- $\delta(r, o(e\ r')) = r'$,
- *Let* $r \in R$ *and* $i(d) \in I'$. *Then*

$$\lambda(r, i(d)) \quad = \quad \begin{cases} i(\xi_r^{-1}(d)) & \text{if } d \in \mathsf{ran}(\xi_r) \text{ and } \xi_r^{-1}(d) \in F(i) \\ i(\perp) & \text{otherwise} \end{cases}$$

Let $r = d\ s \in R$ *and* $o(e\ r') \in O'$. *Let* ι_i *be the valuation that is equal to* ξ_s *if* $d \in \mathsf{ran}(\xi_s)$ *and equal to* $\xi_s \cup \{(\mathsf{in}, d)\}$ *otherwise. Similarly, let* ι_{io} *be the valuation equal to* ι_i *if* $e \in \mathsf{ran}(\iota_i)$ *and equal to* $\iota_i \cup \{(\mathsf{out}, e)\}$ *otherwise. Then* ι_{io} *is injective and* $\mathsf{ran}(\iota_{io}) = \mathsf{ran}(r) \cup \{e\}$. *Suppose* $\mathsf{ran}(r') \subseteq \mathsf{ran}(r) \cup \{e\}$. *Then* $\varrho = \iota_{io}^{-1} \circ \xi_{r'}$ *is well-defined and injective, and*

$$\lambda(r, o(e\ r')) \quad = \quad \begin{cases} (o(\iota_i^{-1}(e)), \varrho) & \text{if } e \in \mathsf{ran}(\iota_i) \\ (o(\perp), \varrho) & \text{otherwise} \end{cases}$$

In the degenerate case $r = \epsilon$ *or* $\mathsf{ran}(r') \not\subseteq \mathsf{ran}(r) \cup \{e\}$, *we define* $\lambda(r, o(e\ r')) = (o(\perp), \emptyset)$.

Example 4.12. *Consider an SUL that behaves as the FIFO-set model of Figure 4.2. As a result of interaction with mapper* $\mathcal{A}_{F_\emptyset}$, *the learner may succeed to construct the abstract hypothesis shown in Figure 4.12. This first hypothesis is incorrect since it*

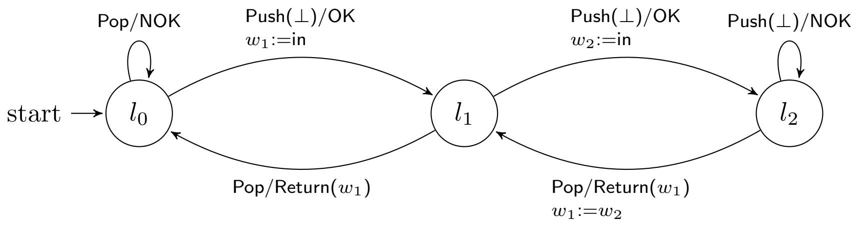


Figure 4.12: First hypothesis for FIFO-set

does not check if the same value is inserted twice. This is because the Abstractor *only generates fresh values during the learning phase. Based on the analysis of a counterexample (to be discussed in the next section), Tomte will discover that it is relevant whether or not the parameter of Push is equal to the value of w_1. Consequently $F($Push$)$ is set to $\{w_1\}$ and Tomte constructs a next hypothesis, for instance the one shown in Figure 4.13. Note that, as the list of memorable values in the initial state is*

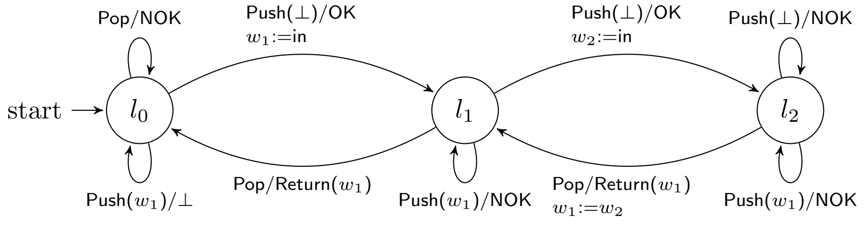


Figure 4.13: Second hypothesis for FIFO-set

empty, there is no concrete action Push(d) that is abstracted to action Push(w_1) in l_0. By the second rule from Definition 4.10, an abstract output $\perp$ is generated in this case.

Theorem 4.5. *Let $\mathcal{R}$ be an input deterministic register automaton with input symbols I and output symbols O such that each location has at most n registers. Let S be the set of collision free neat traces of $\mathcal{R}$, and let $T = \{\mathsf{Oracle}_{\mathcal{R}}(\beta) \mid \beta \in S\}$, that is the set of traces from S in which each output action is annotated with a list of memorable values of the corresponding target state. Let F be a function that assigns to each input symbol a subset of $\{w_0, \ldots, w_{n-1}\}$. Then $U = \alpha_{\mathcal{A}_F}(T)$ is nonempty, prefix closed, input complete and $\equiv_U$ has finite index. If moreover $F(i) = \{w_0, \ldots, w_{n-1}\}$, for all $i \in I$, then U is behavior deterministic.*

In order to show that an hypothesis is incorrect, we first need to concretize it. Using the theory of [10] we get a concretization operator for free, but this concretization operator produces unique-valued register automata in which each output is annotated with the list of memorable values in the target state. Since unique-valuedness leads to a loss of succinctness (and we no longer need the list of memorable values), we have implemented in Tomte an alternative procedure to concretize an abstract deterministic Mealy machine model to a right invariant register automaton:

1. Omit all transitions with output $\perp$ (e.g. the Push(w_1)-loop in location l_0 of Figure 4.13).
2. Whenever, for some location l and input symbol i, there are transitions $l \xrightarrow{i(\perp)/o(d),\varrho} l'$ and $l \xrightarrow{i(w_j)/o(d),\varrho} l'$, then omit the $i(w_j)$-transition (e.g. the Push(w_1)-loop in location l_2 of Figure 4.13; apparently it does not matter whether or not the parameter of Push is equal to the value of w_1).
3. If, for some location l and input symbol i, there are outgoing $i(w)$-transitions for each $w \in W$ then add input guard $\bigwedge_{w \in W} \mathsf{in} \neq w$ to the $i(\perp)$ transition.

4. If a transition has input label $i(w_j)$ then add input guard $\mathsf{in} = w_j$.
5. If a transition has output label $o(\perp)$ then add output guard true.
6. If a transition has output label $o(w_j)$ then add output guard $\mathsf{out} = w_j$.
7. Replace input labels $i(d)$ by i, output labels $o(d)$ by o, and leave all the updates ϱ unchanged.

Example 4.13. *If we apply the above procedure to the Mealy machine of Figure 4.12, then we obtain the register automaton of Figure 4.8 (modulo variable renaming), and if we apply it to the Mealy machine of Figure 4.13, then we obtain the register automaton of Figure 4.2 (again modulo variable renaming).*

In case function F assigns the maximal number of abstract values to each input, the above concretization operator will produce a unique-valued register automaton that is equivalent to the register automaton produced by the concretization operator of [10] (if we forget the lists of memorable values in output actions). In cases where F is not maximal, our concretization operator will typically produce register automata that are not unique-valued. In the next section we will show how, when a flaw in the hypothesis is detected during the hypothesis verification phase, the resulting counterexample can be used for abstraction refinement.

4.8 The Analyzer

During equivalence testing, a test generation component uses the abstract hypothesis to generate abstract test input sequences. This approach allows us to use standard algorithms for FSM conformance testing such as Random Walk or a variation of the W-Method [137]. These test sequences are then concretized, run on both the SUL and the concretized hypothesis, and the resulting outputs are compared. The result is either a concrete counterexample or increased confidence that the hypothesis model conforms to the SUL.

Parameter values in the abstract model can either be $\perp$ or a variable name. If an abstract value is a variable name then the corresponding concrete value is uniquely determined. In contrast, an abstract value $\perp$ allows for infinitely many concretizations and suggests that the SUL behavior is independent of the value picked. By testing we can verify that this is the case. If testing produces a counterexample then this may be used to refine the abstraction and introduce additional abstract values. To more quickly discover such refinements, we test by concretizing $\perp$ to different memorable values.

As example, consider the login model of Figure 4.5. Figure 4.14 depicts the hypothesis built after the first iteration of learning this system. Using the testing approach described, Tomte will eventually find a concrete counterexample trace, say Login(9,9) NOK Register(9,9) OK Register(12,12) NOK Login(9,9) OK. This sequence is a

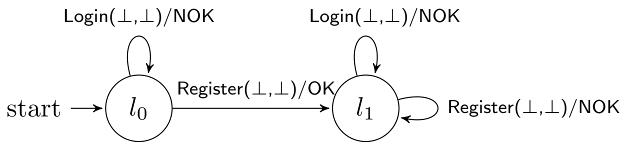


Figure 4.14: Initial abstract hypothesis for login system

valid trace of the SUL but not of the hypothesis, since according to the hypothesis the last output should be NOK. Tomte applies heuristics to reduce the length of the counterexample, in order to simplify subsequent analysis and thus to improve scalability. Two reduction strategies are used: (1) removing loops, and (2) removing single transitions. The first strategy tries to remove parts of the trace that form loops in the hypothesis. These may also form loops in the system and thus not affect the counterexample. The second strategy tries to remove single transitions from the counterexample. The idea behind this is that often different parts of the system are independent of each other, so transitions from the part not causing the counterexample can be removed. Applied to the login case, Tomte first removes loops from the concrete counterexample, which results in the reduced counterexample Register(9,9) OK Login(9,9) OK. Tomte then tries to eliminate each transition, but as the resulting traces do not form counterexamples, this heuristic fails. As a final processing step, the counterexample is made neat, thus becoming Register(0,0) OK Login(0,0) OK. This is done solely to improve the counterexample's readability.

The reduced counterexample is then analyzed by the process depicted in Figure 4.15. The counterexample is first resolved by abstraction refinement. If no refinement can be done, then an abstracted form of the counterexample is sent to the Mealy machine learner, which uses it to further refine the abstract hypothesis.

Abstraction refinement means finding the concrete input parameters that are abstracted to ⊥ but nevertheless form 'relevant' relations with previous parameters. We say that a relation between two parameters is relevant if breaking it by changing a parameter's value also breaks the counterexample. Consequently, the concrete values of these parameters no longer fit ⊥, as they can only take a specific value for the counterexample to hold. Based on relevant relations, we then update the lookahead oracle and construct refined abstractions, that would better fit these parameters. Initially, all parameters values are abstracted to ⊥. This changes as more refined abstractions are created.

A first step to refining is disambiguation, by which any relations between two parameters present that are not relevant for the counterexample, are broken by replacing the latter parameter of the relation with a fresh value. In our running example the trace Register(0,0) OK Login(0,0) OK is changed to Register(0,1) OK Login(0,1) OK, by virtue of the irrelevant equality between the username and password. Breaking relations further would change the observed behavior into one with which the concrete hypothesis would agree.

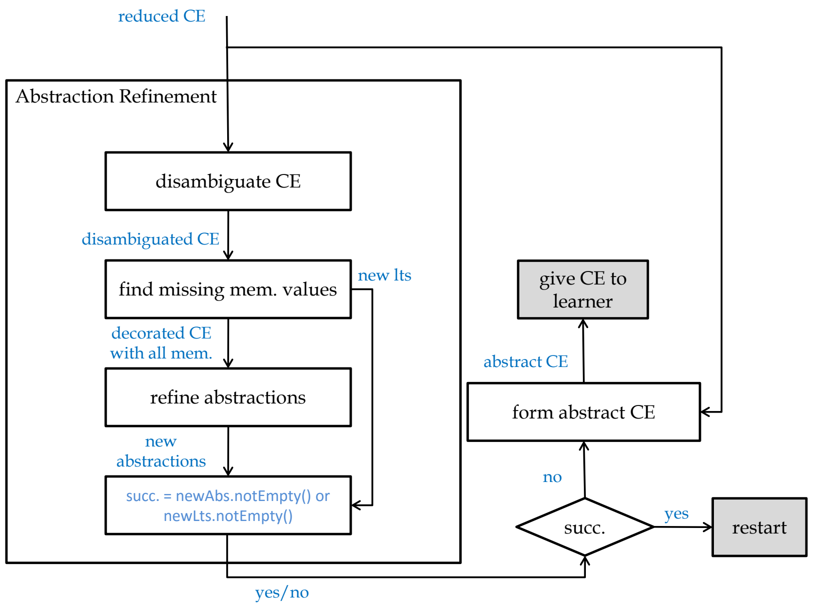


Figure 4.15: Counterexample analysis in Tomte

The disambiguated trace is then sent to the next process, which looks for any missing memorable values and adapts the lookahead oracle so these can all be discovered. The current memorable values are obtained by running the counterexample through the lookahead oracle, which then decorates the trace by placing memorable value lists at the start and after each transition. Such a trace for the login case would be ϵ Register$(0, 1)$ OK ϵ Login$(0, 1)$ OK ϵ. Notice that all the sequences are empty, since initially the lookahead oracle does not find any memorable values. For the last output to be OK, the SUL requires that values 0 and 1 are reused in the Login-input, meaning that the SUL should have remembered them, hence these values should have been found memorable by the lookahead oracle. We say that the lookahead oracle 'misses' these values. In more concrete terms, we say that a parameter value is missing if it is equal to a value from a previous transition, but not contained in the list of memorable values that directly precedes the transition. For the login example, we notice that both 0 and 1 appear as missing values in Login(0,1), since they first emerged in the Register action but they were not included in the memorable set before Login.

The process iterates over the input actions of the decorated trace. Once it passes by an input parameter whose value is judged to be missing, it builds a symbolic lookahead trace that would allow the lookahead oracle to uncover this value. The counterexample is then re-decorated through the augmented lookahead oracle and iteration continues with the next parameter. The end result is a decorated trace which contains no missing values. For the login case, the process updates the lookahead oracle and re-decorates the trace for each of Login's parameters. The end result is the decorated trace where both 0 and 1 are no longer missing: ϵ Register$(0, 1)$ OK $[0, 1]$ Login$(0, 1)$ OK ϵ.

A trace decorated with all memorable values is then sent to the next process, which further decorates the trace so that each concrete value is paired with its corresponding abstract value. This is achieved by running the counterexample through both the mapper (which adds the abstractions) and the lookahead oracle (which adds the memorable values). In the login example, as initially $\perp$ is the only abstract value available, decoration results in the trace

$$\epsilon \ \mathsf{Register}(0 :\perp, 1 :\perp) \ \mathsf{OK} \ [0, 1] \ \mathsf{Login}(0 :\perp, 1 :\perp) \ \mathsf{OK} \ \epsilon.$$

This trace is then iterated and whenever (1) a concrete value is equal to a memorable value, and (2) the corresponding abstraction is $\perp$, a new abstract value is created for the corresponding input symbol and the mapper is updated accordingly. Equality with a memorable value results in an abstraction which simply points to an index in the memorable value list after the previous transition. In the login example, the new abstraction values for the Login-action are w_1 for the first parameter, respectively w_2 for the second, transforming the decorated trace into

$$\epsilon \ \mathsf{Register}(0 :\perp, 1 :\perp) \ \mathsf{OK} \ [0, 1] \ \mathsf{Login}(0 : w_1, 1 : w_2) \ \mathsf{OK} \ \epsilon.$$

The mechanisms of uncovering missing memorable values and new abstractions are closely tied to proper disambiguation of the counterexample. Both these steps consider any equalities between two parameters as relevant to the counterexample. Applying the same process on an ambiguous counterexample might result in resolution of false relations or missing relations which are confounded as was in the login case. Without disambiguation, the counterexample Register$(0, 0)$ OK Login$(0, 0)$ OK would have yielded only one missing value in 0, which would have lead to different refined abstractions. One such abstraction would imply that it is relevant if the second Register parameter is equal to the first, which is clearly not the case.

The final step of counterexample analysis is a simple check if new lookahead traces or new abstract values have been discovered during the last pass. If so, learning is restarted from scratch. Note that memorable values discovered by newly added lookahead traces can have corresponding abstract values which have already been created as a result of a previous refinements. Or the abstract values found might expose relations with previous input values. Similarly, it may happen that the lookahead oracle has already discovered all memorable values, yet for some of these values new abstract values are defined. Learning needs to be restarted as LearnLib currently does not accept on the fly changes to the input alphabet. Moreover, some of the answers to queries from the learning phase might be invalidated by the discovery of new memorable values.

If no new lookahead traces or abstract values have been discovered during a pass, then an abstract version of the counterexample is forwarded to the Mealy machine learner. Obtaining an abstract counterexample involves just running the counterexample through the mapper and lookahead oracle and only collecting the abstracted messages. As an optimization, we also perform this step before abstraction refinement, as it is a considerably cheaper yet just as likely.

According to Figure 4.15, counterexample analysis in Tomte has three possible outcomes: (1) a new lookahead trace is forwarded to the Lookahead Oracle and learning is restarted, (2) a new abstract value is forwarded to the Abstractor and learning is restarted, or (3) a counterexample is forwarded to the learner. By Lemma 4.7, step (1) may only occur a finite number of times. Since the number of input symbols and the number of abstract values are both finite, also step (2) may only occur a finite number of times. If there are no more steps of type (1) or type (2) then, by Theorem 4.5 and Theorem 4.3, the set of abstract traces that can be observed by the learner equals the set of traces of some finite, deterministic Mealy machine. By correctness of the Mealy machine learner, the learner will produce a correct hypothesis after a finite number of queries. Thus we may conclude that our algorithm for learning register automata terminates.

4.9 Evaluation and Comparison

In this section, we compare Tomte 0.41 to other learning tools on a series of benchmarks including the Session Initiation Protocol (SIP), the Alternating Bit Protocol, the Biometric Passport, FIFO-Sets, and a multi-login system. Apart from the last one, all these benchmarks have already been used in [8] for the comparison of Tomte 0.3, a previous version of Tomte, and LearnLib^{RA}. In [5], we compared Tomte 0.4 with LearnLib^{RA} and Tomte 0.3, concluding that Tomte 0.4 performed best in all but two benchmarks. Since then, RALib [54] has been released, a learner building on LearnLib^{RA}, adding several optimizations as well as enabling support for theories other than equality. This made RALib a strong competitor, reporting better numbers for a number of benchmarks. Tomte itself was also improved and can now work with TTT [122], a new and fast algorithm for learning Mealy machines. We focus our evaluation efforts on the more novel Tomte 0.41 and RALib. Readers are referred to [8] and [5] for benchmarking of the 0.3 and 0.4 versions of Tomte and LearnLib^{RA}. Tomte 0.41 generally replicates the numbers obtained by version 0.4 in those benchmarks.

Each experiment consists of learning a simulation of a model implementing a benchmark system or, as in the case of the multi-login system, learning of an actual implementation. Whenever possible we verified the learned model by performing an equivalence check against the simulated model. For the multi-login system we ran a thorough suite of tests. For the FIFO-Set models, we checked the models manually by analyzing the number of states and guards in the learned model.

Tomte 0.41 can now be configured to work with different Mealy machine learners. Traditionally, we have used the Observation Pack algorithm [113], which is enabled in all versions of Tomte. Recently, we have adapted Tomte 0.41 to support the new

TTT algorithm [122]. Similarly, RALib adopts a series of optimizations. We enable all these optimizations apart from the one exploiting parameter typing (unlike in [54]), since all benchmarks used are not typed.

Table 4.1 provides benchmarks for Tomte 0.41 using each of TTT and Observation Pack, and RALib with the optimizations mentioned. Results for each model are obtained by running each learner configuration 10 times with 10 different seeds. Over these runs we collect the average and standard deviation for number of reset queries and inputs applied during learning (denoted **learn res** and **learn inp**), counterexample analysis (denoted **ana res** and **ana inp**) and testing (denoted **test res** and **test inp**). The numbers for testing do not include queries run on the final hypothesis. As RALib does not distinguish counterexample analysis from learning and testing, we exclude statistics for this phase. A final statistic is success (**succ**), denoting for each model the number of successful experiments, that is, experiments which ended with the correct model learned. Since Tomte 0.41 is always successful, we exclude this statistic from its columns.

For consistency, we use the same equivalence oracle across all learners, namely, a random walk oracle configured with a maximum test query length of 100 and an average length of 10, with a maximum of 1000000 tests per equivalence query. The probability of selecting a fresh value is set to 0.1 . We opted for this algorithm, since it was the only algorithm supported RALib. In contrast, Tomte 0.41 can also use more advanced testing algorithms. When learning FIFO-Set 30 we increase the average query length to 100, otherwise testing would most likely fail to find all counterexamples. Similarly, for FIFO-Set 14 we increase it to 50.

We omit running times, as we consider the number of queries to be a superior metric of measuring efficiency, but the reader may find them at [6]. All models apart from the multi-logins and large FIFO-Set models are learned in less than one minute. We limit learning time to 20 minutes.

Results show that TTT significantly brings down the number of learning queries needed by Tomte 0.41, at the cost of more test and counterexample analysis queries. This cost is offset for all but the first model benchmarked. The extent of improvement when we consider the sum of all inputs varies from roughly a 23 % reduction for the SIP model to a factor of 8 reduction for the Palindrome Checker. We also notice that the gap widens with the growing complexity of the models. Furthermore, improvement would likely have been greater had a smarter testing algorithm been used.

RALib beats Tomte 0.41 on several models, particularly SIP and FIFO-Set 7. Unfortunately, its performance is highly erratic, as shown by the high standard deviation. Moreover, RALib is only partially successful at learning some models, while failing completely to learn others. Ultimately, RALib shows promising numbers for some experiments, while for others it seems to suffer a blow up in its algorithm. For the larger models, like the FIFO-Set 30, RALib fails completely.

[6]http://automatalearning.cs.ru.nl/

A cause of RALib's fluctuating performance may be the impact the lengths of counterexamples have on its performance. Long counterexamples may lead to often unnecessarily long suffixes which are very expensive for RALib to process. RALib's underlying algorithm is an adaptation of L^* [19] for register automata which, like L^*, uses an observation table. The performance of such algorithms has been shown to suffer due to long counterexamples. By contrast, Tomte 0.41 leverages more advanced algorithms that more effectively process these counterexamples [122].

The multi-login system benchmark can only be properly handled by Tomte 0.41. The benchmark generalizes the example of Figure 4.3 to multiple users, while adding an additional user ID parameter when logging in and registering. A configurable number of users may register, enabling simultaneous login sessions for different registered users. Tomte 0.41 was able to successfully learn instantiations of multi-login systems for 1, 2 and 3 users. RALib struggled to learn configurations with 1 user, while completely failing for those with more users.

That said, Tomte 0.41's learning algorithm also does not perform nor scale well for higher numbers of users. This can be ascribed to the high number of global abstractions. Such a number is owing to not only the large number of registers, but also to the varying order in which memorable values are found per state.

A memorable value, be it login id or password, can take one index in one state, but another index in a different state. As we use global abstractions, the memorable value would require two distinct abstractions, even though only one is useful in each state. This leads to a large number of abstractions required to cover all indexes memorable values can take.

4.10 Conclusions and Future Work

We have presented a mapper-based algorithm for active learning of register automata that may generate fresh output values. This class is more general than the one studied in previous work [2,7,8,37,55,56]. We have implemented our active learning algorithm in the Tomte tool and have compared the performance of Tomte using each of the Observation Pack or the novel TTT algorithms, to that of RALib on a large set of benchmarks. We measured the total number of inputs required for learning, testing and counterexample analysis. For a set of common benchmarks, TTT helps in significantly bringing down the number of queries used overall. RALib proves competitive but cannot reliably learn all models. In particular, RALib struggles to learn login systems generating fresh passwords. In contrast, Tomte is able to learn models of register automata with fresh outputs, including these systems. Our method for handling fresh outputs is highly efficient and the computational cost of the determinizer is negligible in comparison with the resources needed by the lookahead oracle and the abstractor. Our next step will be an extension of Tomte to a class of models with simple operations on data.

	Tomte 0.41 TTT						Tomte 0.41 OP						RALib				
	learn res	learn inp	test res	test inp	ana res	ana inp	learn res	learn inp	test res	test inp	ana res	ana inp	succ	learn res	learn inp	test res	test inp
Alternating Bit Protocol Sender																	
avg	40	139	24	131	59	172	65	225	8	40	15	29	10/10	824	4621	348	3537
stddev	2	6	6	33	12	57	1	2	3	23	2	4		630	4364	420	4285
Biometric Passport																	
avg	225	924	3156	31304	156	534	729	2883	1791	17899	31	123	0/10	unsuccessful			
stddev	15	70	1927	19354	15	76	1	2	2022	20139	4	28		unsuccessful			
Alternating Bit Protocol Channel																	
avg	20	50	4	12	11	17	37	102	0	0	0	0	10/10	14	25	2	7
stddev	0	1	1	6	3	7	0	0	0	0	0	0		0	1	1	5
Repdigit Palindrome Checker																	
avg	16	15	18	73	29	27	414	815	18	73	30	27	10/10	1172	2301	2200	21814
stddev	0	0	3	38	1	1	0	0	3	38	1	1		113	226	1679	16618
Session Initiation Protocol																	
avg	1619	8757	258	2024	322	1196	2526	13918	115	1083	134	573	10/10	883	4868	40	322
stddev	64	388	77	668	42	178	94	555	41	434	20	98		841	6204	11	112
FIFO-Set(2)																	
avg	37	141	7	17	14	23	53	221	3	14	10	14	10/10	119	719	14	107
stddev	11	49	2	8	1	3	0	0	2	11	0	1		180	1594	7	64
FIFO-Set(7)																	
avg	520	4651	109	983	155	1077	1803	19291	106	1016	139	1018	10/10	435	3112	22	174
stddev	190	1511	30	319	45	422	5	35	38	402	49	577		47	434	24	214
FIFO-Set(14)																	
avg	3214	50924	335	15126	1312	24085	19936	388353	307	14995	974	18978	9/10	2212	27686	6	98
stddev	1189	15714	119	5844	774	17365	25	363	119	5825	520	13317		687	8769	3	110
FIFO-Set(30)																	
avg	25417	815643	96773	9670989	21245	1618024	336175	13276178	96712	9670804	18188	1528670	0/10	unsuccessful			
stddev	10229	257573	48456	4838335	10719	1148441	68	2144	48454	4838462	9051	1103288		unsuccessful			
Multi-Login(1)																	
avg	927	3940	216	1987	203	730	4380	20800	210	2078	125	490	8/10	441371	4398178	170	1719
stddev	42	183	171	1737	19	102	3	15	173	1698	11	69		632641	7740248	131	1376
Multi-Login(2)																	
avg	16756	96291	414	3782	1730	8509	116153	724434	460	4531	891	4503	0/10	unsuccessful			
stddev	3117	19032	145	1447	551	3217	14028	86833	201	1961	211	1338		unsuccessful			
Multi-Login(3)																	
avg	1248119	10371807	134359	1341421	14524	98032	6131225	51443730	9945	99566	2751	17592	0/10	unsuccessful			
stddev	284200	2065284	48533	484092	1820	11391	1064404	9337282	8562	85479	961	7681		unsuccessful			

Table 4.1: Comparison of Tomte 0.41 using Observation Pack and TTT, and RALib

Chapter 5

Learning-Based Testing the Sliding Window Behavior of TCP Implementations

We develop a learning-based testing framework for register automaton models that can express the windowing behavior of TCP, thereby presenting the first significant application of register automata learning to realistic software for a class of automata with Boolean-arithmetic constraints over data values. We have applied our framework to TCP implementations belonging to different operating systems and have found a violation of the TCP specification in Linux and Windows. The violation has been confirmed by Linux developers.

5.1 Introduction

Automata provide both formal and intuitive means of specifying the behavior for a wide range of applications, in particular network protocols. Unfortunately, protocol specifications often are textual and rarely include state machine models. Without such models, it is difficult to test if an application behaves as expected. Manual construction of models is a laborious and error-prone process and models become outdated as soon as the specification changes. Learning-based testing, as sketched in Figure 5.1, alleviates this problem by generating models while testing a system. These models cannot serve as specifications but can be used to check desired properties, which are usually easier to formalize and maintain than complete behavioral models.

Integrating model learning, model-based testing, and model checking allows a tester to automatically obtain a model for a system under test. For a set of test inputs, model learning runs a series of tests on the system until, eventually, it will produce a conjectured model of the system's behavior. This model is used as the basis for model-based testing. Testing can discover counterexamples, which indicate incorrectness of the model. In such case, model learning is restarted, being provided with the counterexample. Once no counterexample is found, the model can be used for checking properties. The output of learning-based testing is threefold: model learning produces a

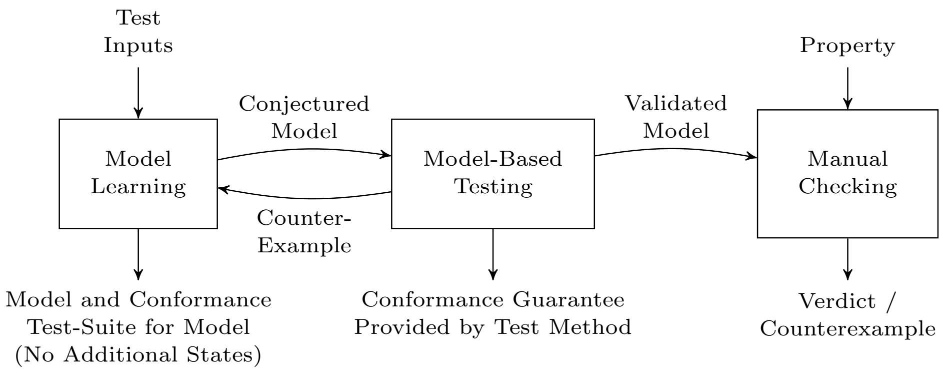


Figure 5.1: Learning-Based Testing with Additional Checking of Properties.

conformance test suite for the model [29], checking of properties can produce examples that document the violation of a specification, and in case no violation is found, testing can yield a conformance guarantee.

In order to instantiate learning-based testing for a certain class of models, one needs a learning algorithm and a testing algorithm for this class of models. In this chapter, we present a learning-based testing framework for a class of register automata that can express the windowing behavior of TCP. Our framework utilizes the SL^* learning algorithm for register automata [58] and a random walk testing algorithm for such register automaton models. The testing algorithm ensures approximate correctness of models with a high confidence. We manually inspect models and find a violation of the TCP specification in Linux and Windows implementations.

Work in this chapter is the first significant application of register automata learning to realistic software for a class of automata with Boolean-arithmetic constraints over data values. Our results show that, on the one hand, learning more expressive models can ease the burden of manually constructed sophisticated test harnesses. On the other hand, experiments show that model learning for more expressive models is very expensive. Future work will focus on scaling learning-based testing to industrial applications as well as on integrating automated model checking into our approach.

Related Work. Learning-based testing in the form that we present here is based on the observation that model learning and model-based testing are merely two sides of the same coin [202]. The term has been introduced in [152] for a combination of model learning, model checking, and random testing. In contrast to work in this chapter, the approach is based on finite state models. On the other hand, model checking is automated and feed the model learning algorithm with counterexamples, leading to higher degree of automation.

Learning-based techniques have been steadily gaining traction for more than a decade, after pioneering work on learning and testing CTI systems [101] and learning and checking systems [170]. Previous applications of learning-based testing or checking have lead to the discovery of flaws in TLS implementations [181] and of various forms of specification non-compliance in TCP [87, 88] and SSH [89] implementations. What all these case studies have in common, is the difficulty of manually constructing a sophisticated test harness for the system. This is in large part caused by the need to abstract away from system functionality, so that the functionality seen by the learner fits within the less expressive formalism the learner can infer, typically mealy machines or DFAs. Our learning setup can infer more expressive register automata, and requires no form of abstraction other than a general one for handling fresh values.

Outline. We provide a brief introduction to TCP in the next section before presenting our learning-based testing framework in Section 5.3. We discuss application of our framework on real TCP implementations in Section 5.4, before concluding in Section 5.5.

5.2 The Sliding Window Behavior of TCP

The Transport Control Protocol (TCP) is a widely used transport layer protocol of the TCP/IP stack, with implementations provided by all operating systems. TCP ensures reliable data transfer between parties. In order to communicate, a TCP client and server application must first establish a TCP connection, which is done by way of a handshake. They can then exchange data over the established connection until one of the parties decides to terminate the connection. A closure procedure ensues, which ultimately removes the connection. In all stages of the protocol, interaction is done by exchanging *TCP segments.* These segments are often the result of calls on the socket interface, which is available to each side and provides access to TCP services. Moreover, each side keeps track of the state of the connection. TCP uses sequence numbers and a sliding receive window to keep track of which segments have been received and acknowledged by the other party. This helps compensate for a potentially lossy communication channel in which reordering of segments can occur (e.g., due to changing routing of segments).

For the sake of exposition, let us assume a setting in which all segments are 1 byte in size. As sequence numbers encode the relative position of a segment in a byte stream, this assumption allows us to confuse the relative position segment in a sequence of segments with its position in a byte stream.

Sequence Numbers. To achieve reliable data transfer, TCP uses sequence and acknowledgement numbers, and flags which are included in the header of all TCP segments. In a stream of segments from a sender to a receiver, the *sequence number* encodes the relative number of a segment in such a stream. The receiver acknowledges a received segment by responding with a segment including as *acknowledgement number* the next expected sequence number. Sequence numbers are generated relative to an Initial Sequence Number (ISN), so the first segment has sequence number ISN,

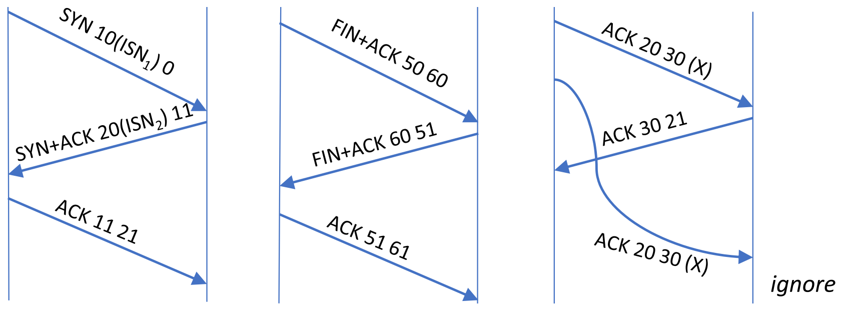


Figure 5.2: TCP handshake, connection closure, and data transfer with re-transmission. Labels show flags, sequence and acknoweldgement numbers. 1 byte of payload marked by (X). Initial Sequence Numbers marked by (ISN).

the second ISN+1... As data is sent, the sequence number increases, as does the acknowledgement number in responses.

Receive Window. Segments received with a sequence number greater than the one expected fall in two categories: those whose sequence number falls within a *receive window* of that expected and those whose sequence number falls outside of the receive window. The former should be processed by the receiver, the latter should be treated as invalid. As a concrete example, only reset segments (segments with the RST flag enabled) with the sequence number within the receive window are processed, and may reset the connection, those whose number lies outside should be ignored. The receive window is included in the TCP header and its value is communicated in each TCP segment a side sends.

Sliding Windows. Once a received segment is successfully processed, the receive window can be moved forward: if a sequence number of a received segment is equal to the sequence number expected, the expected sequence number is increased. If not equal, the expected sequence number is left unchanged. Acknowledgement numbers are also checked. Those equal to the last sequence number sent acknowledge all segments up to this last one. Those greater are unacceptable as they acknowledge segments not yet sent. Those smaller than the last sequence number sent are old acknowledgements. Segments with unacceptable or old acknowledgement numbers are generally discarded.

As stated above, sequence numbers and receive windows are used, among other things, to deal with reordering of routed segments and to prevent the processing of (bytes in) old segments, which are segments carrying already seen data with sequence numbers smaller than the those expected. Old segments are often the result of re-transmissions, which happen when a timeout for receiving an acknowledgement has expired. TCP is full duplex, which means communicating sides maintain two byte streams, one for each direction. Each side keeps track of the next sequence number to be sent, as well as the

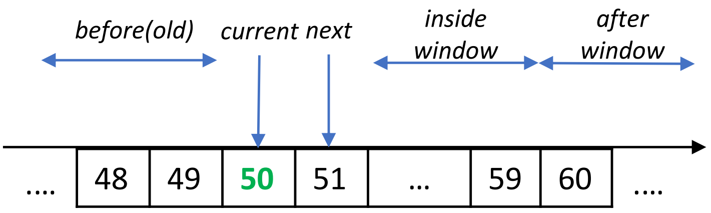


Figure 5.3: Relevant Relations of Sequence Numbers in TCP.

sequence number expected from the other side. To open (via handshake), maintain and close the two byte streams, TCP uses control flags. The SYN flag, for example, marks the beginning of a byte stream, whereas the FIN flag marks the end. Figure 5.2 gives sequence diagrams for typical TCP scenarios.

The description so far assumed that all segments were 1 byte in size. In actuality, the size of a segment is the size of the payload carried, plus 1 if either SYN or FIN flags are enabled, or 0 otherwise. We restrict the learning setting to one where segments carry no payload (thus segments are either of size 0 or 1).

Figure 5.3 depicts the relevant relations sequence numbers may have relative to a current sequence number, in line with our earlier description. These relations are equality and inequality over the current sequence number, and over its summation to one (for segments including either FIN or SYN), and to the receive window size.

5.3 Instantiating Learning-Based Testing for TCP

In order to apply learning-based testing to the windowing behavior of TCP, we instantiate the components of the framework that were sketched in Section 5.1. We use the SL^* active learning algorithm for learning register automaton models [58]. Active learning algorithms rely on the existence of a minimally adequate teacher (cf. [19]) that answers two kinds of queries for the learning algorithm: *output queries* (i.e., execution of tests) and *equivalence queries*. The learning algorithm submits a conjectured model to an equivalence oracle and expects a counterexample to the model (if one exists). In our scenario, we implement this oracle by performing model-based testing on the model.

The SL^* algorithm additionally assumes the existence of a *tree oracle*. A tree oracle produces register automata fragments that encode the relevant data relations for a sequence of actions on a SUT. The resulting setup is shown in Figure 5.4. In order to infer symbolic transitions, e.g., for input $ACK(p_1, p_2)$ with two data parameters $p1$ and p_2 from a state that is reached in the protocol by sending a message $SYN(10, 0)$ and receiving message $SYN + ACK(20, 11)$, the SL^* algorithm will perform a tree query for prefix $SYN(10, 0)$ and suffix ACK. The tree oracle will generate output queries for all relevant concrete instances of ACK messages capturing possible relations

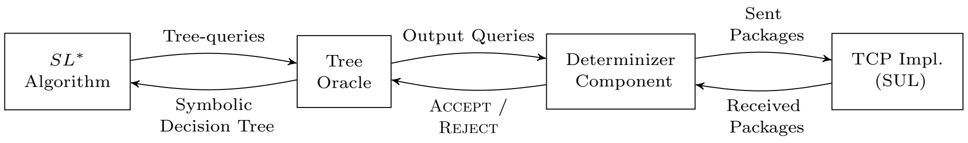


Figure 5.4: Learning Register Automaton Models from Tests.

between values of p_1, p_2 and data values in the prefix (e.g., equality, being a sequence number, or being in a window). The determinizer component will test if output queries are valid traces of a TCP implementation by exchanging actual TCP packages with a system under learning (SUL). The tree oracle encodes the observed behavior and relevant relations as a symbolic decision tree.

In the remainder of this section, we present register automata for the windowing behavior of TCP, tree queries that capture all relevant data relations, and use the presented ideas as a basis for instantiating model-based testing in our framework.

5.3.1 Register Automata

We assume a set Σ of *actions*, each with an arity that determines how many values from $\mathbb{N}$ it takes as parameters (e.g., ACK takes two data values). To simplify presentation, we assume that all actions have arity 1, but it is straightforward to extend to the case where actions have arbitrary arity. A *data symbol* is a term of form $\alpha(d)$, where α is an action and $d \in \mathbb{N}$ is a data value. A *data word* is a sequence of data symbols. The concatenation of two data words w and w' is denoted ww'. In this context, we often refer to w as a *prefix* and w' as a *suffix*. For a data word $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$, let $Acts(w)$ denote its sequence of actions $\alpha_1\ldots\alpha_n$, and $Vals(w)$ its sequence of data values $d_1\ldots d_n$. Let $|w|$ denote the number of symbols in w.

While there are infinitely many data words for every sequence of actions with data parameters, many of these data words are equivalent when considering only relations between data values (e.g., equality, being a sequence number, or being in a window). For a set of relations $\mathcal{R}$, data words $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$ and $w' = \alpha_1(d'_1)\ldots\alpha_n(d'_n)$ are *$\mathcal{R}$-indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if $R(d_{i_1},\ldots,d_{i_j})$ iff $R(d'_{i_1},\ldots,d'_{i_j})$ whenever R is a relation in $\mathcal{R}$ and $i_1,\cdots,i_j$ are indices among $1\ldots n$. We use $[w]_{\mathcal{R}}$ to denote the set of words that are $\mathcal{R}$-indistinguishable from w. A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$.

In order to capture the windowing behavior of TCP, we define the set of relations $\mathcal{R} = \{R_{\otimes,c} \ : \ \otimes \in \{<,\leq,=,\geq,>\} \ \wedge \ c \in \{0,1,100\}\}$, and relation $R_{\otimes,c} \subset \mathbb{N}\times\mathbb{N}$ such that $xR_{\otimes,c}y$ iff $x + c \otimes y$. Relations $R_{\otimes,0}$ encode equality and an order on the sets of sequence numbers. Relations in $R_{\otimes,1}$ encode the successor relation between sequence numbers and $R_{\otimes,100}$ describes windows (of size 100).

We assume a set of *registers* $x_1, x_2, \ldots$ that can store data values of data words. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a formal parameter. An *atomic guard* g over p is a logic formula of form $(x_i + c \otimes p)$ with $\otimes \in \{<, \leq, =, \geq, >\}$ and $c \in \{0, 1, 100\}$. We allow for aggregation of atomic guards into *intervals* of form $(g_1 \wedge g_2)$, where atomic guards g_1 and g_2 specify a lower and an upper bound on p, respectively. A valuation $\nu : \{p, x_1, x_2, \ldots\} \mapsto \mathbb{N}$ satisfies a guard g if $g[\nu] \ = \ g[\nu(p)/p][\nu(x_1)/x_1][\ldots]$ is true and we write $\nu \models g$ in this case.

An *assignment* is a simple parallel update of registers with values from registers or the formal parameter p. We represent an assignment which updates the registers $x_{i_1}, \ldots, x_{i_m}$ with values from the registers $x_{j_1}, \ldots, x_{j_n}$ or p as a mapping π from $\{x_{i_1}, \ldots, x_{i_m}\}$ to $\{x_{j_1}, \ldots, x_{j_n}\} \cup \{p\}$, meaning that the value of the register or parameter $\pi(x_{i_k})$ is assigned to the register x_{i_k}, for $k = 1, \ldots, m$.

Definition 5.1 (Register automaton)**.** *A* register automaton *(RA) is a tuple* $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$*, where*

- L *is a finite set of* locations*, with* $l_0 \in L$ *as the* initial location*,*
- $\mathcal{X}$ *maps each location* $l \in L$ *to a finite set* $\mathcal{X}(l)$ *of registers, and*
- Γ *is a finite set of* transitions*, each of form* $\langle l, \alpha(p), g, \pi, l' \rangle$*, where*
  - $l \in L$ *is a source location,*
  - $l' \in L$ *is a target location,*
  - $\alpha(p)$ *is a parameterized symbol,*
  - g *is a guard over* p *and* $\mathcal{X}(l)$*, and*
  - π *(the* assignment*) is a mapping from* $\mathcal{X}(l')$ *to* $\mathcal{X}(l) \cup \{p\}$*, and*
- λ *maps each* $l \in L$ *to* $\{+, -\}$*.* □

We require register automata to have no initial registers (i.e., $\mathcal{X}(l_0) = \emptyset$) and to be *completely specified* in the sense that for each location $l \in L$ and action α, the disjunction of the guards on the α-transitions from l is equivalent to *true.*

RA Semantics Let us formalize the semantics of RAs. A *state* of an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to $\mathcal{D}$. A *step* of $\mathcal{A}$, denoted $\langle l, \nu \rangle \xrightarrow{\alpha(d)} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ with

- $\nu \ \models \ g[d/p]$, i.e., d satisfies the guard g under the valuation ν, and
- ν' is the updated valuation with $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$.

A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \ldots \alpha(d_n)$ is a sequence of steps of $\mathcal{A}$

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l_1, \nu_1 \rangle \quad \ldots \quad \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l_n, \nu_n \rangle$$

for some initial valuation ν_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by* $\mathcal{A}$ *under* ν_0 if $\mathcal{A}$ has an accepting

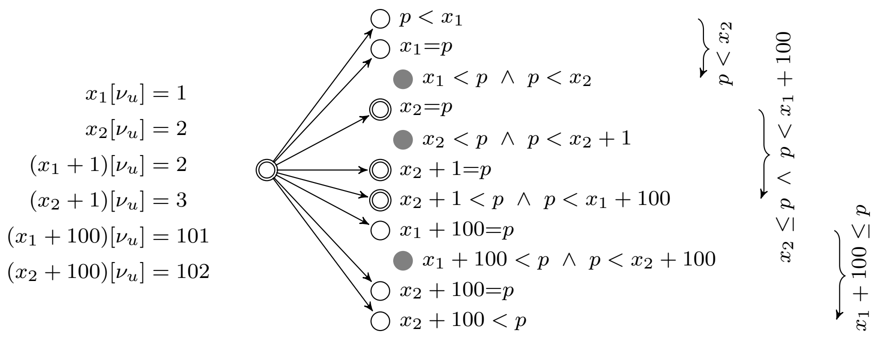


Figure 5.5: Potential (left), maximally refined $(u, \hat{v})$-tree (center), and canonic guards (right) for u with $\nu_u = \{x_1 \mapsto 1, x_2 \mapsto 2\}$ and $\hat{v}$ with $|\hat{v}| = 1$. Actions omitted.

(rejecting) run over w which starts in $\langle l_0, \nu_0 \rangle$. An RA is *determinate* if there is no data word over which it has both accepting and rejecting runs. In this case we interpret an RA $\mathcal{A}$ as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for ACCEPT and $-$ for REJECT. When using register automata as models for reactive system, we refine the set of actions into inputs and outputs (cf. [54]).

5.3.2 Tree Queries

For a data language $\mathcal{L}$, a data word u with $Vals(u) = d_1, \ldots, d_k$, and a set V of sequences of actions (so-called abstract suffixes), a (u, V)*-tree* is a decision tree (a tree-shaped RA) $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ with root l_0 and $\mathcal{X}(l_0) \subseteq \{x_1, \ldots, x_k\}$ that (1) has runs over exactly all data words v with $Acts(v) \in V$ and that (2) accepts a data word v from $\langle l_0, \nu_u \rangle$ iff $uv \in \mathcal{L}$. Please note, that we do not require $\mathcal{X}(l_0)$ to be empty for decision trees and let ν_u such that $\nu_u(x_i) = d_i$ for $x_i \in \mathcal{X}(l_0)$ and d_i the i-th data value of u.

A tree oracle for $\mathcal{L}$ is a function $\mathcal{O}$ that for any prefix u and set of abstract suffixes V constructs a (u, V)-tree $\mathcal{O}(u, V)$. In other words, $\mathcal{O}(u, V)$ is the tree oracle's answer to a tree query with prefix u and abstract suffixes V. The SL^* algorithm systematically poses tree queries to a tree oracle, combining resulting symbolic decision trees (SDTs) into a conjectured model. We can implement a tree oracle by starting with a maximally refined symbolic decision tree that has one unique sequence of transitions for every $\mathcal{R}$-indistinguishable class of words $[uv]_{\mathcal{R}}$ with $Acts(v) \in V$ and then compute a more concise tree by iteratively merging equivalent subtrees.

Maximally refined SDTs. For simplicity, we describe the generation of a maximally refined symbolic decision tree for a prefix u and a single abstract suffix $\hat{v}$. This allows us to omit actions from the presentation. For $|Vals(u)| = k$, the *potential of* u is the set of terms $(x_i + c)$ with $1 \le i \le k$ and $c \in \{0, 1, 100\}$ that can appear in guards after u. The valuation ν_u (with $\nu_u(x_i) = d_i$ for $d_i \in Vals(u)$) induces an order on the

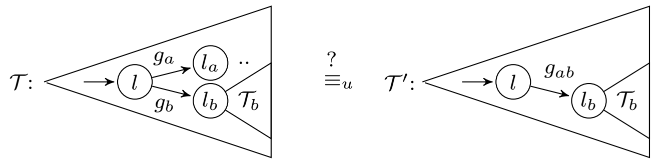


Figure 5.6: Merging Sub-Trees of an SDT.

terms in the potential. An example of this order is shown on the left of Figure 5.5 for a word u with two data values.

Omitting the trivial case of the empty sequence, let $|\hat{v}| = 1$ for the moment. We generate guards for cases p smaller than the smallest term in the potential of u, p equal to one of the terms, p in the interval between two successive terms, and p greater than any term in the potential of u. These guards are maximally refined: each (satisfiable) guard describes one class $[uv]_{\mathcal{R}}$ of $\mathcal{R}$-indistinguishable words. We instantiate each guard with the help of a constraint solver and use an output query to determine if $uv \in \mathcal{L}$. Figure 5.5 (middle) exemplifies the construction. As indicated by gray lines on the left of the figure, some terms in the potential are equal. For these cases we pick one of the equal terms as the basis for guards. Gray colored guards cannot be instantiated and are omitted.

In the general case of $|\hat{v}| > 1$, we apply the above technique iteratively, generating sequences of guards and transitions for the parameters of $\hat{v}$. We maintain data values of the suffix symbolically during sequence generation and only instantiate complete sequences of guards. The approach scales to sets of suffix sequences as we construct maximally refined paths: paths of suffixes with common prefixes will have common guards for those prefixes and can be expressed as trees.

Maximally abstract SDTs and Monotonicity. In order to guarantee convergence of learning on a canonical automaton, the SL^* makes some monotonicity requirements on tree oracles [58]. For growing sets of abstract suffixes V, $V', \dots$ with $V \subset V'$, it has to be shown that $\mathcal{O}(u, V')$ refines $\mathcal{O}(u, V)$ by only adding registers to $\mathcal{X}(l_0)$, and only refining guards of transitions. Additionally, if decision trees $\mathcal{O}(u, V)$ and $\mathcal{O}(u', V)$ cannot be made equal under some renaming of registers from $\mathcal{X}(l_0)$ in one tree, trees $\mathcal{O}(u, V')$ and $\mathcal{O}(u', V')$ cannot become equal either by such a renaming. These conditions trivially hold on maximally refined SDTs. Unfortunately, however, maximally refined SDTs do not lead to finite models during learning as the shape of a tree depends on the length of the prefix. We transform maximally refined SDTs into more abstract trees by merging transitions and equivalent sub-trees (akin to BDD minimization), thereby hiding irrelevant structural differences between trees.

The essential idea is that two (u, V)-trees $\mathcal{T}$ and $\mathcal{T}'$ are semantically equivalent after u, denoted by $\mathcal{T} \equiv_u \mathcal{T}'$, if both trees accept the same set of suffixes under initial valuation ν_u with $\nu_u(x_i) = d_i$ for $d_i \in Vals(u)$. We can check semantic equivalence with finitely many test runs (i.e., one for each path in a maximally refined SDT for

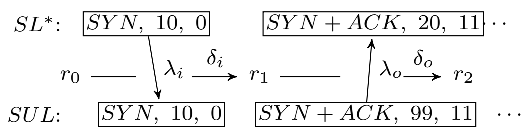


Figure 5.7: Translation between Neat Trace and SUL Trace.

V). Let now l be a location in $\mathcal{T}$ with outgoing transitions to l_a and l_b, guarded by g_a and g_b, respectively, as sketched in Figure 5.6. For some new guard g_{ab}, equivalent to $(g_a \vee g_b)$, we construct $\mathcal{T}'$ from $\mathcal{T}$ w.l.o.g. by removing the transition from l to l_a and the sub-tree rooted at l_a. On the transition from l to l_b, we replace g_b by g_{ab} (cf. right part of the figure). We abstract g_a and g_b into g_{ab} if $\mathcal{T} \equiv_u \mathcal{T}'$.

In order to arrive at a canonical representation, we perform merging in a fixed order: we always merge guards for the smallest possible terms with respect to the order on the potential (cf. maximally refined trees). This ensures that merging always results in intervals. An example is shown on the right of Figure 5.5. Merged guards are obtained from top (smaller terms) to bottom (greater terms).

Our semantic merging process satisfies all three requirements: Adding more suffixes (and hence paths) cannot lead to merging subtrees that could not be merged before. Guards are refined into finer intervals. Since the original boundaries will be maintained, monotonic growth of registers follows. Finally, since abstract trees are semantically equivalent to maximally refined trees, differences between trees are preserved when adding suffixes.

Output queries observe the behavior of the SUL on a sequence of test inputs. In learning-based testing, these queries are computed by executing tests on the actual system under test.

Testing has to be done in an adaptive fashion, synchronizing data values that are used in test inputs by the learning algorithm and those used in actual tests as the SUL may introduce new sequence numbers during tests. As an example, the learning algorithm may assume to receive a message $SYN + ACK$ with (new) sequence number 1. Then, in the actual communication the SUL sends a random new sequence number.

To tackle this problem, the work [5] introduces a determinizer component, placed between the learner and the SUL. This component provides the learner with a deterministic, or 'neat' view of the SUL, by constructing and applying a 1 to 1 mapping from regular values to *neat values*. This mapping transforms all relation equivalent traces (input/output sequences) encountered to a single neat trace. The learner then infers the SUL only in terms of its neat traces.

Output Queries.

We extend the determinizer concept to a setting with inequalities and sums. Our definition focuses on data values and ignores actions, which are invariant under mapping. The determinizer is the mapper $\mathcal{D} = \langle R, r_0, \delta_i, \delta_o, \lambda_i, \lambda_o \rangle$ over states $R = \{r \subseteq \mathbb{N} \times \mathbb{N} \mid r \text{ finite and one-to-one}\}$ with initial state $r_0 = \emptyset$. Value transformations (λ) and mapper updates (δ) are defined for $c \in 0, 1, 100$ and $x, y, n, m \in \mathbb{N}$ as follows.

$$\lambda_i(r, n) = \begin{cases} x + c & \text{if } m + c = n \text{ for some } (x, m) \in r \\ \mathsf{smaller}(\mathsf{dom}(r)) & \text{if } m + c > n \text{ for all } (\cdot, m) \in r \\ \mathsf{fresh}(\mathsf{dom}(r)) & \text{if } m + c < n \text{ for all } (\cdot, m) \in r \\ (x + y)/2 & \text{else; for } (x - c_1, m_l - c_1), (y - c_2, m_u - c_2) \in r \\ & \text{s.t. } (m_l < n < m_u) \text{ and } (m_u - m_l) \text{ minimal} \end{cases}$$

$$\lambda_o(r, x) = \begin{cases} n + c & \text{if } y + c = x \text{ for some } (y, n) \in r \\ \mathsf{fresh}(\mathsf{ran}(r)) & \text{otherwise} \end{cases}$$

$$\delta_i(r, n) = \begin{cases} r & \text{if } (\cdot, n) \in r \\ r \cup \{\ (\lambda_i(r, n),\ n)\ \} & \text{otherwise} \end{cases}$$

$$\delta_o(r, x) = \begin{cases} r & \text{if } (x, \cdot) \in r \\ r \cup \{\ (x,\ \lambda_o(r, x))\ \} & \text{otherwise} \end{cases}$$

There, dom and ran denote domain and image of a function. Functions fresh : $\mathbb{N}^* \to \mathbb{N}$ and smaller : $\mathbb{N}^* \to \mathbb{N}$ generate fresh values and smaller values. For $X \subset \mathbb{N}$ we use the concrete functions $\mathsf{fresh}(X) := (\lfloor max(X)) \div s_u \rfloor + 1) \times s_u$ and $\mathsf{smaller}(X) := (\lfloor \mathsf{min}(X) \div s_l \rfloor - 1) \times s_l$. Step sizes s_u and s_l are fixed big enough to avoid collisions (accidental relations between data values) during experiments.

Figure 5.7 shows an example application of the mapper, producing a neat trace from Figure 5.2. Whenever the system generates an output, the determinizer processes it by replacing the output values with neat values before delivering the output to the learner. Conversely, on generating a concrete input, the learner passes it to the determinizer which replaces neat input values with regular values, and sends the resulting input to the SUL. Every time it processes a value, the determinizer updates its state.

5.3.3 Model-Based Testing

We instantiate the testing part of our framework with a relative simple adaptation of a random algorithm to the scenario of register automaton models. For a register automaton model $\mathcal{A}$, each test run begins by traversing the model to a randomly selected location of $\mathcal{A}$ and is continued by a random sequence of inputs until either a discrepancy is discovered between model and system under test, or until the run terminates and a new run starts.

Our extension consists in selecting data values for inputs. For a run with current prefix w and next input α, we use the machinery introduced above (the potential of a word, and symbolic guards that describe classes $[w\alpha(d)]_{\mathcal{R}}$ of data words) as a basis for computing a pool of data values for α. The pool contains one data value d for each $\mathcal{R}$-indistinguishable class $[w\alpha(d)]_{\mathcal{R}}$ of data words. We add a bias to the selection of data values, so that values in or related to those stored in registers in $\mathcal{A}$ after running over w are more likely to be picked.

We can easily obtain a PAC-inspired conformance guarantee (cf. [206]) with this testing method for the probability distribution on the set of data words induced by a model $\mathcal{A}$ and the above strategy for selecting tests. With respect to this distribution, $\mathcal{A}$ is an ϵ-approximation of SUL if $\sum_{w \in S} Pr(w) \leq \epsilon$ for the symmetric difference S of sets of words accepted by $\mathcal{A}$ and SUL. The probability of $\mathcal{A}$ not being an ϵ-approximation of the SUL after performing k independent test runs is at most $(1-\epsilon)^k$. For some confidence value δ, we simply choose k such that $(1-\epsilon)^k < \delta$ (i.e., such that $k > ln(\delta)/ln(1-\epsilon)$.

5.4 Testing TCP Implementations

We have implemented the theories introduced earlier into RALib [54]. We then set up an experimental setup through which we could connect RALib to various TCP clients. RALib inferred models, which we checked manually for conformance with the specification.

5.4.1 Experimental Setup

The experimental setup used to learn TCP is similar to the setup used in [87] and [88]. As in those works, the alphabet used to learn TCP defines two types of inputs. The first type is *packet inputs*, used to describe TCP segments sent to the system. These inputs are parameterized by TCP flag combinations, sequence and acknowledgement numbers. The second type of inputs is *socket inputs* such as `connect` and `close`, referring to the methods defined by the socket interface. Outputs defined are *packet outputs*, which bear the same structure as packet inputs and describe TCP segments generated by the system, and *timeouts*, which suggest that no output was generated by the system. For model learning, we use the SL^* algorithm with the theory and optimizations discussed earlier. Additionally, we used techniques for reducing the size of counterexamples as shorter counterexamples tend to lead to shorter suffixes, which greatly decreases the number of inputs needed to run. For sample techniques and a corresponding discussion we refer to [131]. Model-based testing was done using the algorithm described in the previous section. Finally, to speed up learning, we used multiple SUL instances in parallel. In particular during testing, tests were distributed and run evenly among the instances. We could reliably use up to 3 instances in parallel. Above that number, we encountered instances of missed responses. This could be improved upon with a more efficient setup.

5.4.2 Experiments and Results

We attempted to learn TCP client implementations of Linux, FreeBSD and Windows. We chose clients, since they are simpler to learn and contain less redundancy compared to servers (cf. [88]). In terms of the configurations used, we disabled adaptive receive windows (or window scale), so that receive windows remain fixed over the course of each test. Moreover, in the segments sent to the SUL we advertise the same receive

Table 5.1: Learning Statistics. BASE stands for Baseline. [T] marks Use of Typing.

SUL	Alpha.	Term.	Inp. Loc.	Num. Hyp.	Learning Inputs	Learning Resets	Testing Inputs	Testing Resets
Linux 3.19	`[T]BASE`	yes	6	15	4,311	947	113,921	11,720
	`BASE`	yes	6	15	9,930	2,168	116,479	12,339
	`[T]BASE+ACK`	yes	8	21	77,922	13,414	119,768	12,289
FreeBSD 11.0	`[T]BASE`	yes	6	16	4,239	933	113,953	11,708
	`BASE`	yes	6	16	9,958	2,152	116,446	12,333
	`[T]BASE+ACK`	no	8	21	418,977	80,200	81,024	8,367
Windows 10	`BASE-CLOSE`	no	6	14	193,712	24,848	119,768	12,289

window as that of the SUL. Doing so we avoid having to include an additional sum constant for our own receive window.

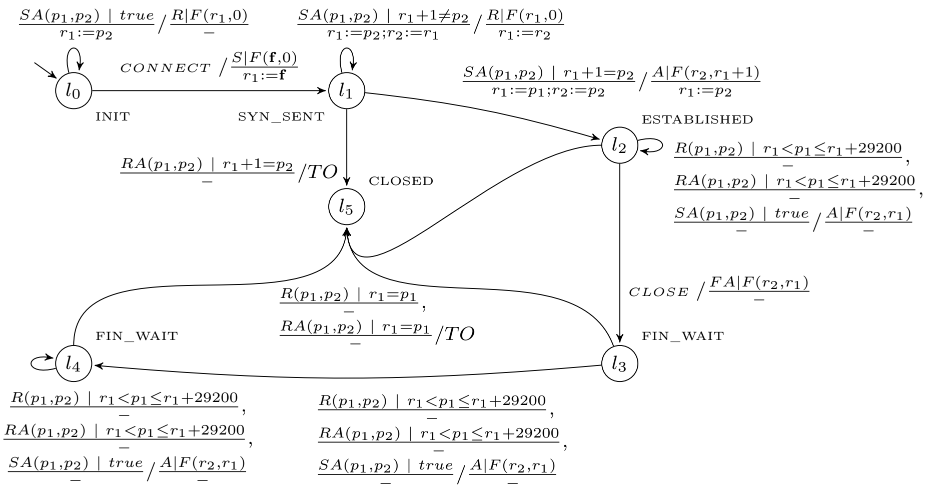
Our baseline alphabet consists of the `connect`, `SYN+ACK`, `ACK+RST`, `RST` and `close` inputs. This alphabet covers several states in the specification. The alphabet should also reveal how SULs in these states react to RST segments. These segments are generated in cases where one side abruptly terminates a connection and should be processed only if their sequence numbers are in window of the expected. We have also extended the alphabet with the `ACK` input if learning with the baseline was successful. To obtain models in an adequate time, we do not explore data relations between all formal parameters in some experiments. This optimization has been introduced as *typing* of symbolic parameters in [54].

Once a hypothesis was constructed, we tested it using the algorithm presented earlier. We have set the size of the random sequence to 10 (sufficient for exploring the behavior we are interested in) and ran $15,000$ tests on the final hypothesis. Using the confidence metric from the previous section, this yields a confidence of more than $99,9\%$ that a model is an 0.05%-approximation of the SUL for data words up to a length of 10 — relative to the probability distribution our randomized testing algorithm generates over the set of data words.

Table 5.1 reports the setting, termination status and learning statistics for all experiments done. The setting indicates the concrete SUL, the alphabet relative to the baseline and whether typing was used. Successful experiments took at most two days to complete, the determining factors being the size in parameters of the suffixes and the 0.3 seconds wait time used for each response before concluding a timeout. We automatically terminated experiments still unresolved after $500,000$ inputs. For these experiments, we still display the last hypothesis and learning numbers at the point of termination. Results are available on RALib's website.[1]

Using both un-typed and typed baseline alphabets we inferred models for Linux and FreeBSD. We inferred a model for Linux using the `ACK`-extended typed alphabet, but not for BSD. Learning FreeBSD for this setting followed a similar course to learning Linux, leading to a similar hypothesis. Testing generated a counterexample, whose

[1]See: https://goo.gl/23VNfv



$\frac{SA(p_1,p_2)\ |\ true}{r_1:=p_2} / \frac{R|F(r_1,0)}{-}$
$\frac{SA(p_1,p_2)\ |\ r_1+1\neq p_2}{r_1:=p_2;r_2:=r_1} / \frac{R|F(r_1,0)}{r_1:=r_2}$
$CONNECT / \frac{S|F(\mathbf{f},0)}{r_1:=\mathbf{f}}$
$\frac{SA(p_1,p_2)\ |\ r_1+1=p_2}{r_1:=p_1;r_2:=p_2} / \frac{A|F(r_2,r_1+1)}{r_1:=p_2}$
l_0
INIT
l_1
SYN_SENT
ESTABLISHED
l_2
$\frac{R(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{RA(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{SA(p_1,p_2)\ |\ true}{-} / \frac{A|F(r_2,r_1)}{-}$
$\frac{RA(p_1,p_2)\ |\ r_1+1=p_2}{-} /TO$
CLOSED
l_5
$CLOSE / \frac{FA|F(r_2,r_1)}{-}$
$\frac{R(p_1,p_2)\ |\ r_1=p_1}{-}$, $\frac{RA(p_1,p_2)\ |\ r_1=p_1}{-} /TO$
FIN_WAIT
l_4
FIN_WAIT
l_3
$\frac{R(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{RA(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{SA(p_1,p_2)\ |\ true}{-} / \frac{A|F(r_2,r_1)}{-}$
$\frac{R(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{RA(p_1,p_2)\ |\ r_1<p_1\leq r_1+29200}{-}$, $\frac{SA(p_1,p_2)\ |\ true}{-} / \frac{A|F(r_2,r_1)}{-}$

Figure 5.8: Model of Linux Client. Flags are replaced by their starting characters (i.e. FIN by F, SYN by S). TO denotes a timeout, **f** denotes a fresh value. We group inputs with guards soliciting the same output and assignment over registers and use input/output notation. Inputs have guards over parameters. In outputs, parameters are instantiated.

processing resulted in a long new suffix. The suffix proved too expensive for tree queries to terminate within the input bounds set.

We couldn't learn Windows models even after removing the `CLOSE` input. Analysis of the last conjectured model and the generated tests revealed behavior inconsistent with the specification: Windows accepts sequence numbers up to and including window size plus one in the `ESTABLISHED` state for `RST` inputs. This helps demonstrate a limitation of our approach: relevant data relations $\mathcal{R}$ are an input to learning and convergence is guaranteed only for systems that respect $\mathcal{R}$ (cf. Section 5.3.1).

5.4.3 Analysis of Conformance to RFC

Before reflecting upon the models learned, we mention a bug discovered while conducting experiments. In our attempt to learn the TCP Linux client with an alphabet comprising the baseline plus the `ACK+FIN` packet, we noticed that, while in the `FIN_WAIT` state, the Linux client upon receiving an `ACK+FIN` segment with an invalid acknowledgement number, would still process and acknowledge the FIN flag. This would have shown up in the learned model, unfortunately, poor scalability meant we could not learn a model for this setup. The bug was reported and subsequently fixed[2].

[2]See: https://www.spinics.net/lists/netdev/msg436743.html

```
#define after(seq2, seq1)  before(seq1, seq2)
static inline bool before(__u32 seq1, __u32 seq2) {
        return (__s32)(seq1-seq2) < 0;
}
static inline bool tcp_sequence(
    const struct tcp_sock *tp, u32 seq, u32 end_seq) {
  return !before(end_seq, tp->rcv_wup) &&
    !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
}
```

Listing 5.9: Relevant Code of TCP Implementation in Linux Kernel.

Figure 5.8 presents the model learned for Linux using the baseline alphabet. The models learned for FreeBSD and Linux are near identical with one exception. Linux defines an in-window sequence number as a value up to and including $rcv.nxt + win$ (for a next expected sequence number $rcv.nxt$). FreeBSD excludes the upper bound. Windows, on the other hand, even seems to include $rcv.nxt + win + 1$. RFC 793 [176, page 26] specifies that an in-window sequence smaller is strictly smaller than $rcv.nxt + win$. Thus, FreeBSD conforms to the upper bound requirement whereas Linux and Windows do not. For Linux, we trace this violation to code in the most recent kernel, v4.11.[3] Listing 5.9 shows the relevant code snippets. To check whether a sequence number is not after the window, they use the $!(seq > rcv.nxt + win)$ conjunct, allowing $rcv.nxt + win$ to be within the window. We inquired Linux developers about this issue and they confirmed it and said they would issue a fix for it. During our experiments, we have uncovered a different, unrelated, bug relating to faulty re-transmissions for which a fix has been issued.

Aside from that, reset processing seems to be implemented as stated in the RFC with the remark that both systems implement the 'Blind Reset Attack Using RST Bit' safe guard introduced in RFC 5961 [179, page 7], by which only RST segments with the sequence number equal to the expected sequence number cause the termination of a connection. RST segments whose sequence number is in window but not equal to the expected sequence number prompt a 'challenge ACK response'. We can verify that this is the case by analyzing the Linux model's responses to RST segments in the `ESTABLISHED` and `FIN_WAIT1` states. As a note, RFC 5961 might have been the cause of the inconsistency remarked previously. As of this writing, RFC 5961 gives a wrong description of the within/outside window conditions of RFC 793. The error had been reported in 2016 and is included in the RFC errata[4].

5.5 Conclusion

Work in this chapter introduces the first application of register automata learning to real networked systems, in the form of TCP clients. To that end, we have developed

[3]See: https://goo.gl/9A8ZYM

[4]See: https://www.rfc-editor.org/errata/rfc5961

the theories needed to learn TCP into the learning framework of [58]. We implemented heuristics that improve scalability of learning and developed a component that deals with non-determinism in fresh data values. The application of our learning-based testing setup resulted in models for TCP client implementations of Linux and FreeBSD. Our setup helped reveal violation of the RFC 793 standard [176] in Linux and Windows. In Linux we identified the root cause for the violation in the Kernel code.

In a next step, we plan to produce models for extended sets of inputs and models of TCP servers. Despite the optimizations used, we eventually faced combinatorial blow up in the number of required tests. Combining learning with static or symbolic analysis methods may help reducing this blow up by identifying more precisely the relations one should test for. This will also address the limitation of fixed relations that prevented us from learning a model for Windows.

Chapter 6

Model Learning as a Satisfiability Modulo Theories Problem

We explore an approach to model learning that is based on using *satisfiability modulo theories* (SMT) solvers. To that end, we explain how DFAs, Mealy machines and register automata, and observations of their behavior can be encoded as logic formulas. An SMT solver is then tasked with finding an assignment for such a formula, from which we can extract an automaton of minimal size. We provide an implementation of this approach which we use to conduct experiments on a series of benchmarks. These experiments address both the scalability of the approach and its performance relative to existing active learning tools.

6.1 Introduction

We are interested in algorithms that construct black-box state diagram models of software and hardware systems by observing their behavior and performing experiments. Developing such algorithms is a fundamental research problem that has been widely studied. Roughly speaking, two approaches have been pursued in the literature: *passive learning* techniques, where models are constructed from (sets of) runs of the system, and *active learning* techniques, that accomplish their task by actively doing experiments on the system.

Gold [96] showed that the passive learning problem of finding a minimal DFA that is compatible with a finite set of positive and negative examples, is NP-hard. In spite of these hardness results, many DFA identification algorithms have been developed over time, see [108] for an overview. Some of the most successful approaches translate the DFA identification problem to well-known computationally hard problems, such as SAT [107], vertex coloring [91], or SMT [161], and then use existing solvers for those problems.

Angluin [19] presented an efficient algorithm for active learning a regular language L, which assumes a *minimally adequate teacher* (MAT) that answers two types of queries about L. With a *membership query*, the algorithm asks whether or not a given word w is in L, and with an *equivalence query* it asks whether or not the language L_H of a hypothesized DFA H is equal to L. If L_H and L are different, a word in the symmetric difference of the two languages is returned. Angluin's algorithm has been successfully adapted for learning models of real-world software and hardware systems [170,178,205], as shown in Figure 6.1. A membership query (MQ) is implemented by bringing the

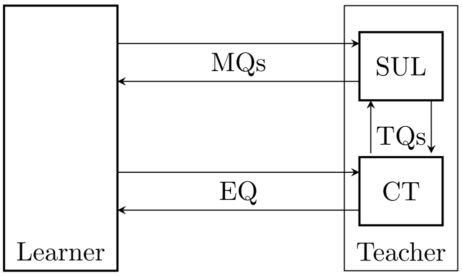


Figure 6.1: Model learning within the MAT framework.

system under learning (SUL) in its initial state and the observing the outputs generated in response to a given input sequence, and an equivalence query (EQ) is approximated using a *conformance testing tool* (CT) [137] via a finite number of *test queries* (TQ). If these test queries do not reveal a difference in the behavior of a hypothesis H and the SUL, then we assume the hypothesis model is correct.

Walkinshaw et al. [213] observed that from each passive learning algorithm one can trivially construct an active learning algorithm that only poses equivalence queries. Starting from the empty set of examples, the passive algorithm constructs a first hypothesis H_1 that is forwarded to the conformance tester. The first counterexample w_1 of the conformance tester is then used to construct a second hypothesis H_2. Next counterexamples w_1 and w_2 are used to construct hypothesis H_3, and so on, until no more counterexamples are found.

In this chapter, we compare the performance of existing active learning algorithms with passive learning algorithms that are 'activated' via the trick of Walkinshaw et al. [213]. At first, this may sound like a crazy thing to do: why would one compare an efficient active learning algorithm, polynomial in the size of the unknown state machine, with an algorithm that makes a possibly superpolynomial number of calls [20] to a solver for an NP-hard problem? The main reason is that in practical applications i/o interactions often take a significant amount of time. In [184], for instance, a case study of an interventional X-ray system is described in which a single i/o interaction may take several seconds. Therefore, the main bottleneck in these applications is the total number of membership and test queries, rather than the time required to decide which queries to perform. Also, in practical applications the state machines are often small, with at most a few dozen states (see for instance [3,11,184]). Therefore, even though passive learning algorithms do not scale well, there is hope that they can still handle these applications. Active learning algorithms rely on asking a large number of membership queries to construct hypotheses. Passive learning algorithms pose no membership queries, but instead need a larger number of equivalence queries, which are then approximated using test queries. A priori, it is not clear which approach performs best in terms of the total number of membership and test queries needed to learn a model.

Our experiments compare the original L* [19] and the state-of-the-art TTT [122] active learning algorithm with an SMT-based passive learning algorithm on a number of practical benchmarks. We encode the question whether there exists a state machine with n states that is consistent with a set of observations into a logic formula, and then use the Z3 SMT solver [76] to decide whether this formula is satisfiable. By iteratively incrementing the number of states we can find a minimal state machine consistent with the observations. As equivalence oracle we use a state-of-the-art conformance testing algorithm based on adaptive distinguishing sequences [136, 191]. In line with our expectations, the passive learning approach is competitive with the active learning algorithms in terms of the number of membership and test queries needed for learning.

An advantage of SMT encodings, when compared for instance with encodings based on SAT or vertex coloring, is the expressivity of the underlying logic. In recent years, much progress has been made in extending active learning algorithms to richer classes of models, such as register automata [5, 58, 115] in which data may be tested and stored in registers. We show that the problem of finding a register automaton that is consistent with a set of observations can be expressed as an SMT problem, and compare the performance of the resulting learning algorithm with that of Tomte [5], a tool for active learning of register automata, on some simple benchmarks. New algorithms for active learning of FSMs, Mealy machines and various types of register automata are often extremely complex, and building tools implementations often takes years [5, 58, 122]. Adapting these tools to slightly different scenarios is typically a nightmare. One such scenario is when the system is missing *reset* functionality. This renders most active learning tools impractical, as these rely on the ability to reset the system. Developing SMT-based learning algorithms in settings with and without resets only took us a few weeks. This shows that the SMT-approach can be quite effective as a means for prototyping learning algorithms in various settings.

The rest of this chapter is structured as follows. Section 6.2 describes how one can encode the problem of learning a minimal consistent automaton in SMT. The scalability and effectiveness of our approach, and its applicability in practice are assessed in Section 6.3. Conclusions are presented in Section 6.4.

6.2 Model Learning as an SMT Problem

This section describes how to express this problem in a logic formula. *If and only if* there exists an assignment to the variables of this formula that makes it true, then exists an automaton A with at most n states that is consistent with S. We use an SMT solver to find such an assignment. If the SMT solver concludes that the formula is satisfiable, then its solution provides us with A.

We distinguish three types of *constraints*:

- *axioms* must be satisfied for A to behave as intended by its definition.
- *observation constraints* must be satisfied for A to be consistent with S.
- *size constraints* must be satisfied for A to have n states or less.

Hence, the problem can be solved by iteratively incrementing n until the encoding of the axioms, observation constraints and size constraints is satisfiable.

In the following subsections, we present encodings for deterministic finite automata (Section 6.2.1 and Section 6.2.2), Moore and Mealy machines (Section 6.2.3), register automata (Section 6.2.4), and input-output register automata (Section 6.2.5). Extensions for all automata without registers also appear in a preliminary version of this work [192].

6.2.1 An Encoding for Deterministic Finite Automata

A *deterministic finite automaton* (DFA) accepts and rejects *strings*, which are sequences of labels. We define a DFA as follows:

Definition 6.1. *A DFA is a tuple* (L, Q, q_0, δ, F)*, where*

- L *is a finite set of* labels*,*
- Q *is a finite set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*
- $\delta : Q \times L \to Q$ *is a* transition function *for states and labels,*
- $F \subseteq Q$ *is a set of* accepting states*.*

Let x be a string. We use x_i to denote ith label of x. We use $x_{[i,j]}$ to denote the substring of x starting at position i and ending at position j (inclusive), i.e. $x = x_{[1,|x|]}$.

A DFA A accepts a string if its computation ends in an accepting state. This can be formalized as follows. Let $x \in L^*$ be a string, then A accepts x if a sequence of states $q'_0 \dots q'_{|x|}$ exists such that

1. $q'_0 = q_0$,
2. $q'_i = \delta(q'_{i-1}, x_i)$ for $1 \leq i \leq |x|$, and

3. $q'_{|x|} \in F$.

Let S_+ be a set of strings that should be accepted, and let S_- be a disjoint set of strings that should be rejected. Let S be the set that contains all of these strings, along with their labels, i.e. $S = \{(x, \texttt{true}) : x \in S_+\} \cup \{(x, \texttt{false}) : x \in S_-\}$. A DFA is *consistent* with S if it accepts all strings in S_+, and rejects all strings in S_-.

This leads us to a natural encoding for finding a consistent DFA in satisfiability modulo the theories of inequality and uninterpreted functions. We encode a DFA as follows:

- Q is a finite subset of the (non-negative) natural numbers $\mathbb{N}$,
- $q_0 = 0$,
- The set of accepting states F is encoded as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$.

The following size constraint ensures that A has at most n states:

$$\forall q \in \{0, \ldots, n-1\} \quad \forall l \in L \quad \bigvee_{q'=0}^{n-1} \delta(q, l) = q' \tag{6.1}$$

If we assume without loss of generality that the initial state is 0, then we can add the following constraints for the strings in S_+:

$$\forall x \in S_+ \quad \lambda(\ \delta(\ldots \delta(\delta(0, x_1), x_2), \ldots x_{|x|})\ ) = \texttt{true} \tag{6.2}$$

Similarly, we can add the following constraints for the strings in S_-:

$$\forall x \in S_- \quad \lambda(\ \delta(\ldots \delta(\delta(0, x_1), x_2), \ldots x_{|x|})\ ) = \texttt{false} \tag{6.3}$$

6.2.2 A Better Encoding for Deterministic Finite Automata

The nesting in the set of constraints given by Equation 6.2 and Equation 6.3 might lead to many redundant constraints for the theory solver. To give an example, if two strings share a non-empty prefix, the prefix is encoded twice, once for each string. One solution is to define the constraints implied by strings in a non-nested way. Similarly to Heule and Verwer [107], and Bruynooghe et. al. [49], we use an *observation tree* (OT) for this. This can be considered a partial, tree-shaped automaton that is *exactly consistent* with S, i.e. it accepts only the set S_+ and rejects only the set S_-. We define an OT for a set of labeled strings in Definition 6.2.

Definition 6.2. *An OT for a set of strings $S = \{S_+, S_-\}$ is a tuple (L, Q, λ), where*

- *L is a set of labels,*
- *$Q = \{x \in L^* : x$ is a prefix of a string in $S_+ \cup S_-\}$,*
- *$\lambda : S_+ \cup S_- \to \mathbb{B}$ is a output function for the strings, with $x \in S_+ \iff \lambda(x) = \texttt{true}$.*

Now, let us explain how one can construct a set of constraints for finding a DFA $A = (L, Q^A, q_0, \delta^A, F)$ that is consistent with an OT $T = (L, Q^T, \lambda^T)$ for a given set $S = \{S_+, S_-\}$. Let us (again) consider the set of states Q^A as a set of non-negative integers with $q_0 = 0$, and let us encode the set of accepting states F as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$. Recall that a DFA is consistent if and only if it accepts all strings in S_+ and rejects all strings in S_-, i.e. for each x in S $\lambda^A(\delta^A(q_0, x)) = \lambda^T(x)$ (we slightly abuse notation here by extending $\delta^A : Q \times L^* \to Q$ to strings). Such a DFA has at most as many states as the OT (but typically significantly less). Therefore, there must exist a surjective (i.e. many-to-one) function from the strings of the OT to states of the DFA:

$$map : Q^T \to Q^A \tag{6.4}$$

Our goal is to find a set of constraints for *map* that make sure that our target DFA A is consistent. For this we define the following observation constraints:

$$map(\epsilon) = q_0 \tag{6.5}$$
$$\forall xl \in Q^T : x \in L^*, l \in L \quad \delta^A(map(x), l) = map(xl) \tag{6.6}$$
$$\forall x \in S_+ \cup S_- \quad \lambda^A(map(x)) = \lambda^T(x) \tag{6.7}$$

Equation 6.5 maps the empty string to the initial state of A. Equation 6.6 encodes the observed prefixes as transitions of A while Equation 6.7 encodes the observed outputs, with λ^A encoding F.

To meet the minimality requirement, we are interested in finding the 'smallest' *map* function; i.e. there should be no function with a smaller image that satisfies these constraints. For this purpose we can re-use one of the size constraints presented earlier (Equation 6.2).

6.2.3 Adaptations for Moore and Mealy Machines

An advantage of the encoding presented in Section 6.2.2 (as opposed to the one presented in Section 6.2.1) is that it can easily be modified to learn *transducers*. Transducers are automata that generate output strings. As such, they can be used to model input-output behaviour of software.

A *Moore machine* is a transducer that generates an output label initially and each time it (re-) enters a state. We define a Moore machine in Definition 6.3.

Definition 6.3. *A Moore machine is a tuple $(I, O, Q, q_0, \delta, \lambda)$, where*

- *I is a finite set of* input labels*,*
- *O is a finite set of* output labels*,*
- *Q, q_0 and δ are a set of states, the initial state, and a transition function respectively, and*
- *$\lambda : Q \to O$ is a output function that maps states to output labels.*

A set of observations S for a Moore machine consists of *traces*, which are pairs (x^I, x^O) where $x^I \in I^*$ is an *input string* and $x^O \in O^*$ is an *output string* with $|x^O| = |x^I| + 1$. A Moore machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

A *Mealy machine* is a transducer that generates an output label each time it makes a transition. We define a Mealy machine in Definition 6.4.

Definition 6.4. *A Mealy machine is a tuple* $(I, O, Q, q_0, \delta, \lambda)$*, where*

- I, O, Q, q_0 *and* δ *are the same as for a Moore machine (Definition 6.3), and*
- $\lambda : Q \times I \to O$ *is a output function that maps transitions to output labels.*

A set of observations S for a Mealy machine consists of traces (x^I, x^O) where $x^I \in I^*$ is an input string and $x^O \in O^*$ is an output string with $|x^O| = |x^I|$. Similarly to a Moore machine, a Mealy machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

It has been shown that Moore and Mealy machines are equi-expressive if we neglect the initial output label generated by a Moore machine (see e.g. [110]). Therefore, we can define an OT for a set S of traces for a Moore or Mealy machine $A = (I, O, Q^A, q_0, \delta^A, \lambda^A)$ in a similar way. We choose to define such an *input-output observation tree* (IOOT) as follows.

Definition 6.5. *An IOOT for a set of traces* S *is a tuple* (I, O, Q, λ)*, where*

- I *and* O *are sets of input labels and output labels respectively,*
- $Q = \{x \in I^* : x$ *is a prefix of an input string of a trace in* $S\}$*,*
- $\lambda : Q \to O$ *is a output function with* $\lambda(x^I_{[0,i]}) = x^O_i$ *for all* $(x^I, x^O) \in S$ *and* $1 \leq i \leq |x^I|$.

Observe that λ is defined for all states. Also, observe that there is no need for λ to be a transition output function for Mealy machines, because there is only one string that ends in each state of an IOOT.

Let $T = (I, O, Q^T, \lambda^T)$ be an IOOT for a set of traces S, then we can determine if there is a Moore or Mealy machine A with at most n states that is consistent with S by using the set of constraints and axioms from Section 6.2.2, if we replace Equation 6.7 with Equation 6.8 (Moore machines) or Equation 6.9 (Mealy machines).

$$\forall x \in Q^T \quad \lambda^A(map(x)) = \lambda^T(x) \tag{6.8}$$

$$\forall xl \in Q^T : x \in I^*,\ l \in I \quad \lambda^A(map(x), l) = \lambda^T(xl) \tag{6.9}$$

6.2.4 An Encoding for Register Automata

DFAs and Mealy machines typically do not scale well if the domain of inputs, or the domain of data parameters for inputs, is large. The reason for this is that the semantics of the data parameters are modeled implicitly using states and transitions; inputs with different parameters are simply regarded as different inputs. A better solution is to use a richer formalism that can model them more efficiently and exploit the resulting symmetries in the state space.

A *register automaton* (RA) is such a formalism. An RA can be seen as an automaton that is extended with a set of *registers* that can store data parameters. The values in these registers can then be used to express conditions over the transitions of the automaton, or *guards*. If the guard is satisfied the transition is fired, possibly storing the provided data parameter (this is called an *assignment*) and bringing the automaton from the current *location* to the next. As such, an RA can be used to accept or reject sequences of label-value pairs. In contrast to automata without memory, the "states" in a register automaton are called locations because the *state* of the automaton also comprises the values of the registers. Therefore, an exponential number of possible states can be modeled using a small number of locations and registers.

The RAs that we define here have the following restrictions:

right invariance Transitions do not imply (in) equality of distinct registers.

non-swapping Values are never moved from one register to another.

unique values Registers always store unique values.

The first two restrictions are inherent to the definition used, the third is necessary to avoid the non-determinism caused by two used registers holding the same value. While these restrictions may cause a blow-up in the number of states required to be consistent with a set of action strings [53], it has been shown that they do not affect expressivity [6, Theorem 1], i.e. for any register automaton that does not have these restrictions, there exists an equivalent register automaton in the class that we are concerned with. For a formal treatment of these restrictions and their implications, we refer to [5] and [53].

We define an RA as follows.

Definition 6.6. *An RA is a tuple* $(L, R, Q, q_0, \delta, \lambda, \tau, \pi)$*, where*

- L*,* Q*,* q_0 *and* λ *are a set of labels, a set of locations, the start location, and a location output function respectively,*
- R *is a finite set of* registers*,*
- $\delta : Q \times L \times (R \cup \{r_\perp\}) \to Q$ *is a* register transition function*,*
- $\tau : Q \times R \to \mathbb{B}$ *is a* register use predicate*, and*
- $\pi : Q \times L \to (R \cup \{r_\perp\})$ *is a* register update function*.*

We call a label-value pair an *action* and denote it $l(v)$ for input label l and parameter v. We assume without loss of generality that parameter values are integers ($\mathbb{Z}$). A sequence of actions is called an *action string*, and is denoted by σ. A set of observations S for an RA consists of action strings that should be accepted S_+, and a set of action strings that should be rejected S_-. An RA is consistent with $S = \{S_+, S_-\}$ if it accepts all action strings in S_+, and rejects all action strings in S_-.

Formally, an RA can be considered as a DFA (Definition 6.1) enriched with a finite set of registers R and two additional functions. The first function, τ, specifies which registers are in use in a location. In a location q there can be two types of transitions for a label l and parameter value v:

- If the value v is equal to some used register r, then the transition $\delta(q, l, r)$ is taken.
- Else (if the value v is different to all used registers), the *fresh* transition $\delta(q, l, r_\perp)$ is taken.

The second function, π, specifies if and where to store a value v when this fresh transition $(\delta(q, l, r_\perp))$ is taken:

- If $\pi(q, l) = r_\perp$ then the value v on transition $\delta(q, l, r_\perp)$ is not stored.
- Else (if $\pi(q, l) = r$ for some register $r \in R$), the value v on transition $\delta(q, l, r_\perp)$ is stored in register r.

Let us describe the axioms that we need for the RA to behave as intended. First, we require that no registers are used in the initial location:

$$\forall r \in R \quad \tau(q_0, r) = \texttt{false} \tag{6.10}$$

Second, if a register is used after a transition, it means that it was used before, or it was updated:

$$\begin{aligned} &\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \forall r' \in (R \cup \{r_\perp\}) \\ &\quad \tau(\delta(q, l, r'), r) = \texttt{true} \implies (\ \tau(q, r) = \texttt{true} \lor (r' = r_\perp \land \pi(q, l) = r)\ ) \end{aligned} \tag{6.11}$$

Third, if a register is updated, then it is used afterwards:

$$\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \pi(q, l) = r \implies \tau(\delta(q, l, r_\perp), r) = \texttt{true} \tag{6.12}$$

Our goal is to learn an RA that is consistent with a set of action strings $S = \{S_+, S_-\}$. For this, we need to define a function that keeps track of the valuation of registers during runs over these action strings. Let $A = (L, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A)$ be an RA, and let $T = (L \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for S. In addition to the *map* function (Equation 6.4), we define a *valuation function val* that maps a state of T and a register of A to the value that it contains:

$$val : Q^T \times R^A \to \mathbb{Z} \tag{6.13}$$

Before we construct constraints for the action strings, we *determinize* them by making them *neat* [10, Definition 7]. An action string is *neat* if each parameter value is either equal to a previous value, or equal to the largest preceding value plus one. Let a be a parameterized input, and let a(3)a(1)a(3)a(45) be an action string, then a(0)a(1)a(0)a(2) is its corresponding neat action string, for example. Aarts et al. show that in order to learn the behavior of a register automaton it suffices to study its neat action strings, since any other action string can be obtained from a neat one via a zero respecting automorphism [10, Section 5].

Constructing constraints for an RA is a bit more involving than for the formalisms that we have discussed so far. First, we map empty string to the initial location of A (Equation 6.5). Second, we assert that a register is updated if its valuation changes, and that it is not updated if it keeps its value:

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A$$
$$\mathit{val}(\sigma l(v), r) \neq \mathit{val}(\sigma, r) \implies \pi^A(\mathit{map}(\sigma), l) = r \tag{6.14}$$

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A$$
$$\mathit{val}(\sigma l(v), r) = \mathit{val}(\sigma, r) \implies \pi^A(\mathit{map}(\sigma), l) \neq r \tag{6.15}$$

Additionally, we assert the inverse (i.e. that a register's valuation changes if and only if it is updated):

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A$$
$$\mathit{val}(\sigma l(v), r) = \begin{cases} v & \text{if } \delta^A(\mathit{map}(\sigma), l, r_\perp) = \mathit{map}(\sigma l(v)) \\ & \quad \wedge\ \pi(\mathit{map}(\sigma), l) = r \\ \mathit{val}(\sigma, r) & \text{otherwise} \end{cases} \tag{6.16}$$

Third, we encode the observed transitions:

$$\forall \sigma l(v) \in Q^T$$
$$\mathit{map}(\sigma l(v)) = \begin{cases} \delta^A(\mathit{map}(\sigma), l, r) & \text{if } \exists! r \in R : \tau^A(\mathit{map}(\sigma), r) = \texttt{true} \\ & \qquad \wedge\ \mathit{val}(\sigma, r) = v \\ \delta^A(\mathit{map}(\sigma), l, r_\perp) & \text{otherwise} \end{cases} \tag{6.17}$$

Finally, we encode the observed outputs. This can be done in the same way as for DFAs (see Equation 6.7).

The task for the SMT solver is to find a solution that is consistent with these constraints. Obviously, we are interested in an RA with the minimal number of locations and registers. The number of locations can be limited in the same way as states were limited for DFAs (see Equation 6.1). The number of registers is defined by the variables r that we quantify over in the presented equations. Therefore, they can be limited as such. In our case, the number of registers is never higher than the number of locations (because we can only update a single register from each location). Hence, the learning problem can be solved iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

6.2.5 An Extension for Input-Output Register Automata

An *input-output register automaton* (IORA) is a register automaton transducer that generates an output action (i.e. label and value) after each input action. As in the RA-case, we restrict both input and output labels to a single parameter. Input and output values may update registers. Input values may be tested for (dis-)equality with values in registers. Output values can be equal to the stored values, or may be fresh. As such, an input-output register automaton can be used for modeling software that produces parameterized outputs.

For a formal description of IORAs we refer to [10]. We define an IORA in Definition 6.7. Again, in the interest of our encoding, our definition is very different from that in [10]. Despite this, the semantics are similar.

Definition 6.7. *An IORA is a tuple* $(I, O, R, Q, q_0, \delta, \lambda, \tau, \pi, \omega)$*, where*

- I *and* O *are finite, disjoint sets of input and output labels,*
- R*,* Q*,* q_0*,* τ *and* π *are the same as for an RA (Definition 6.6),*
- $\delta : (Q \cup \{q_\perp\}) \times (I \cup O) \times (R \cup \{r_\perp\}) \rightarrow (Q \cup \{q_\perp\})$ *is a* register transition function with a sink location*,*
- $\lambda : (Q \cup \{q_\perp\}) \rightarrow \mathbb{B}$ *is a* location output function with a sink location*, and*
- $\omega : Q \rightarrow \mathbb{B}$ *is a* location type function *that returns* `true` *if a location is an* input location*, and* `false` *if it is an* output location*.*

A set of observations S for an IORA consists of *action traces*, which are pairs (σ^I, σ^O) where $\sigma^I \in (I \times \mathbb{Z})^*$ is an *input action string*, and $\sigma^O \in (O \times \mathbb{Z})^*$ is an *output action string* with $|\sigma^I| = |\sigma^O|$. An IORA is consistent with a set S if for each pair $(\sigma^I, \sigma^O) \in S$ it generates σ^O when provided with σ^I.

Despite that semantically an IORA is a transducer, we define it as an RA (Definition 6.6) which distinguishes between input and output labels, and which defines an additional function ω for the location type. From an *input location* transitions are allowed only for input actions. After an input action the IORA reaches an *output location*, in which a *single* transition is allowed. This transition determines the output action generated in response, as well as the input location the IORA will transition to. Transitions that are not allowed lead to a designated *sink location*, which is denoted $q_\perp$.

Using this definition allows us to incorporate the axioms defined for our RA encoding (Equations 6.10–6.12) also in our IORA encoding. To these, we add the following axioms for an IORA to behave as intended.

First, observe that we do not use λ as an output function for an IORA. Instead, we use it to denote which locations are allowed. Hence, we require that the sink location $q_\perp$ is the only rejecting location:

$$\forall q \in (Q \cup \{q_\perp\}) \quad \lambda(q) = \begin{cases} \texttt{false} & \text{if } q = q_\perp \\ \texttt{true} & \text{otherwise} \end{cases} \tag{6.18}$$

Second, we require that transitions do not lead to the sink location:

$$\forall q \in Q \quad \forall o \in O \quad \forall r \in (R \cup \{r_\bot\}) \quad \omega(q) = \texttt{true} \implies \delta(q, o, r) = q_\bot \tag{6.19}$$

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\bot\}) \quad \omega(q) = \texttt{false} \implies \delta(q, i, r) = q_\bot \tag{6.20}$$

$$\forall l \in I \cup O \quad \forall r \in (R \cup \{r_\bot\}) \quad \delta(q_\bot, l, r) = q_\bot \tag{6.21}$$

Finally, we require that input locations are *input enabled* (Equation 6.22), and that there is only one transition possible in an output location (Equation 6.23):

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\bot\}) \quad \omega(q) = \texttt{true} \implies \delta(q, i, r) \neq q_\bot \tag{6.22}$$

$$\forall q \in Q \quad \exists! o \in O \quad \exists! r \in (R \cup \{r_\bot\}) \quad \omega(q) = \texttt{false} \implies \delta(q, o, r) \neq q_\bot \tag{6.23}$$

Our goal is to learn an IORA $A = (I, O, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A, \omega^A)$ that is consistent with a set of action traces S. Because of the nature of our encoding, we consider each action trace $\sigma = (\sigma^I, \sigma^O)$ in S as an interleaving of the input action string σ^I and the output action string σ^O, i.e. $\sigma = \sigma^I_1 \sigma^O_1 \dots \sigma^I_{|\sigma^I|} \sigma^O_{|\sigma^I|}$. Let $T = ((I \cup O) \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for such strings.

The constraints for an IORA can now be constructed in the same way as for an RA (Equation 6.5 and Equation 6.14–6.17). Observe that we do not use λ to encode the observed outputs (this is already done by encoding the transitions of the OT). Instead, λ is used to denote which locations are allowed. All the locations in Q are allowed (because we have observed them) and $q_\bot$ is the only location that is not allowed ($\lambda(q_\bot) = \texttt{false}$ by Equation 6.18). As such, we add the following constraint:

$$\forall \sigma \in Q^T \quad map(\sigma) \neq q_\bot \tag{6.24}$$

We can now determine if there is an IORA with at most n locations and m registers in the same way as for RAs, i.e. by iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

6.3 Implementation and Evaluation

We implemented our encodings using Z3Py, the Python front-end of Z3 [76][1]. Our tool can generate an automaton model from a given set of observations (passive learning), or a reference to the system and a tester implementation (active learning), also when this system cannot be reset. We have also implemented a tester for the classes of automata supported. The tester generates test queries (or *tests*) each test consisting of an access sequence to an arbitrary state in the current hypothesis, and a sequence generated by a random walk from that state. In experiments, we configure the tester to build shorter tests. Longer tests worsen the scalability of our tool, and are unneeded

[1]See https://gitlab.science.ru.nl/rick/z3gi/tree/lata

for learning small models. All experimental results shown were obtained using our most efficient encodings, namely, those involving an OT and not relying on linear equalities (all other encodings performed considerably worse in terms of scalability).Validity of the learned models was ensured by running a large number of tests on the last hypothesis and checking the number of states. We conducted a series of experiments to assess the scalability and effectiveness of our approach.

Our first experiment assesses the scalability of our encodings by adapting the scalable *Login*, *FIFO set* and *Stack* benchmarks of [5] to DFAs, Mealy machines, RAs and IORAs. The systems benchmarked are IORA by nature, and are parameterizable by their size, which refers to either the maximum number of registered users, or to the size of the collection. The systems only generate `ok` and `nok` labels as output, joined by no parameters. This facilitates the generation of RA/Mealy/DFA representations by applying an adapter over the system, exposing an interface corresponding to the respective formalism. The RA adapter, for example, accepts a sequence of inputs only if all outputs generated by the system are `ok`, otherwise it rejects the sequence. Our adaptation made the Mealy and DFA versions of FIFO set and Stack systems equivalent, hence we only consider FIFO sets for these formalisms. The Login systems are simplified by removing the password parameter (so `login`, `register` and `logout` are done solely by supplying a user id), as our implementation does not yet support actions with multiple parameters. To generate tests, we used the testing algorithm described earlier. The maximum length of the random sequence is $3 + size$, where $size$ is the number of users or elements in the system. The *solver timeout* – the amount of time the solver was provided to compute a solution or indicate its absence – was set to 10 seconds for the DFA and Mealy systems, and to 10 minutes for the RA and IORA systems whose constraints could take considerably longer to process. We initially terminated learning runs whenever the SMT solver failed to return a result within this time bound (or the SMT solver *timed out*). We then realized that even in cases where the SMT solver timed out, it might still find a solution in a subsequent iteration (for a greater n). This solution might not be minimal, but it was nevertheless consistent with past observations. We thus allowed each learning run to iterate until an upper bound was reached. For each system we performed 5 learning runs and collated the resulting statistics.

Results are shown in Table 6.1. Columns describe the system, the number of successful learning runs, the number of states/locations (which may vary due to loss of minimality) and registers (where applicable), average and standard deviation for the number of tests and inputs used in learning except for validating the last hypothesis, and for the amount of time learning took. The table only includes entries for systems we could learn.

In our second experiment we used our tool to learn simulated models obtained by the learning case studies described in [3, 11, 184]. These models are Mealy machines detailing aspects of the behavior of bankcard protocols, biometric passports and power control services (PCS). For the purpose of this experiment we connected the

Table 6.1: Scalable experiments

Model	succ	states	regs	tests		inputs		time(sec)	
		loc		avg	std	avg	std	avg	std
DFA_FIFOSet(1)	5	3.0		17.0	8.28	53.0	36.35	0.44	0.07
DFA_FIFOSet(2)	5	4.0		19.0	10.33	80.0	45.58	0.68	0.11
DFA_FIFOSet(3)	5	5.0		28.0	12.71	141.0	69.99	1.83	0.46
DFA_FIFOSet(4)	5	6.0		48.0	41.12	294.0	293.18	3.44	0.42
DFA_FIFOSet(5)	5	7.0		108.0	19.62	788.0	161.23	7.24	1.54
DFA_FIFOSet(6)	5	8.0		125.0	42.06	953.0	361.76	22.34	9.03
DFA_FIFOSet(7)	5	9.0		136.0	34.52	1126.0	344.18	28.55	12.65
DFA_FIFOSet(8)	5	10.0		228.0	81.11	2156.0	832.02	76.28	42.49
DFA_FIFOSet(9)	2	11.5		413.5	161.93	4194.0	2104.35	199.99	26.1
DFA_Login(1)	5	4.0		100.0	30.8	432.0	140.46	3.06	0.52
DFA_Login(2)	5	7.0		167.0	95.4	932.0	618.15	14.94	2.54
DFA_Login(3)	5	11.0		446.0	100.18	3092.0	781.73	131.84	39.26
Mealy_FIFOSet(1)	5	2.0		4.0	1.1	14.0	5.12	0.13	0.0
Mealy_FIFOSet(2)	5	3.0		9.0	2.51	39.0	12.54	0.49	0.09
Mealy_FIFOSet(3)	5	4.0		16.0	5.32	90.0	30.96	0.71	0.07
Mealy_FIFOSet(4)	5	5.0		14.0	8.64	108.0	62.35	1.44	0.64
Mealy_FIFOSet(5)	5	6.0		24.0	11.03	166.0	86.08	1.96	0.27
Mealy_FIFOSet(6)	5	7.0		36.0	14.85	307.0	151.59	3.81	0.8
Mealy_FIFOSet(7)	5	8.0		41.0	14.38	373.0	138.2	9.7	2.57
Mealy_FIFOSet(8)	5	9.0		90.0	26.53	928.0	310.48	20.87	2.73
Mealy_FIFOSet(9)	5	10.0		131.0	22.68	1547.0	296.28	34.64	5.67
Mealy_FIFOSet(10)	5	11.0		162.0	66.45	1948.0	787.28	60.08	12.93
Mealy_FIFOSet(11)	5	12.0		280.0	110.65	3694.0	1722.57	79.75	23.69
Mealy_FIFOSet(12)	4	15.5		370.0	200.12	5021.0	3312.85	227.15	290.99
Mealy_FIFOSet(13)	2	14.5		526.5	318.91	8021.5	5608.06	190.36	51.96
Mealy_Login(1)	5	3.0		12.0	5.45	52.0	21.43	0.78	0.07
Mealy_Login(2)	5	6.0		44.0	12.15	264.0	83.27	6.37	1.09
Mealy_Login(3)	5	10.0		104.0	10.03	726.0	69.93	52.4	4.83
Mealy_Login(4)	1	16.0		241.0	0.0	2094.0	0.0	370.19	0.0
RA_Stack(1)	5	3.0	1	32.0	21.18	109.0	90.48	3.18	0.86
RA_Stack(2)	5	5.0	2	202.0	71.88	1018.0	394.58	124.72	53.41
RA_FIFOSet(1)	5	3.0	1	49.0	12.92	180.0	52.56	4.94	6.07
RA_FIFOSet(2)	5	6.0	2	365.0	88.41	2025.0	578.33	333.09	334.12
RA_Login(1)	5	4.0	1	306.0	163.18	1336.0	765.23	54.96	9.99
RA_Login(2)	3	8.0	2	1606.0	345.22	9579.0	2163.67	6258.11	1179.27
IORA_Stack(1)	5	2.0	1	7.0	1.58	24.0	6.63	8.77	1.92
IORA_FIFOSet(1)	5	2.0	1	8.0	3.27	31.0	9.36	6.45	0.84
IORA_Login(1)	5	3.0	1	33.0	6.65	152.0	29.89	1509.18	477.04

open-source tester used in [191][2]. This produces tests similar to our own, but extended by distinguishing sequences. These tests are parameterized by both the length of the random sequence and a factor k. We set both the length and the k factor to 1. We note that our simple tester (which doesn't append distinguishing sequences) could not reliably found counterexamples for several of these models. We attribute this to the large size of the models' input alphabets. The solver timeout was set to 1 minute.

Table 6.2: Case-study experiments

Model	succ	states	alpha	tests		inputs		time(sec)	
			size	avg	std	avg	std	avg	std
Biometric Passport	5	6	9	173.0	90.75	848.0	574.57	28.85	3.82
MAESTRO	5	6	14	1159.0	280.78	6193.0	1690.15	330.87	15.33
MasterCard	5	6	14	703.0	192.03	3560.0	1133.96	337.44	80.17
PIN	5	6	14	767.0	188.76	3825.0	1095.39	328.0	41.85
SecureCode	5	4	14	290.0	67.33	1340.0	318.29	82.25	29.46
VISA	5	9	14	839.0	169.53	5005.0	1161.63	1933.03	498.98
PCS_1	5	8	9	704.0	178.94	3861.0	1123.0	201.74	19.68
PCS_2	5	3	9	72.0	7.96	284.0	22.01	8.89	1.3
PCS_3	5	7	9	555.0	175.55	2973.0	1078.96	146.89	21.89
PCS_4	5	7	9	583.0	224.07	3029.0	1626.2	158.33	18.66
PCS_5	5	9	9	1158.0	163.37	6218.0	1165.34	750.83	135.16
PCS_6	5	9	9	778.0	517.2	4087.0	3204.23	735.55	75.77

Results are shown in Table 6.2. Columns are as in the previous experiment, with an additional column used to describe the size of the input alphabet. Our approach is able to learn all models, though it takes a considerable amount of time for the larger models. There are no cases where we cannot learn the model.

Our third experiment pits our approach against LearnLib (v0.12.1) [119] and Tomte (v0.41) [5]. LearnLib is a known FSM learning framework, while Tomte is a learner for IORAs. Both LearnLib and Tomte are configured to use TTT, a state-of-the-art learning algorithm within Angluin's framework. LearnLib is additionally configured to use the original L* learning algorithm. The setups for all learners use caching to ensure that only tests uncovering new observations are included in statistics. We compare our approach to LearnLib on both the scalable and case study models, and to Tomte on the scalable models. The testers are the same as in previous experiments. Due to the high standard deviation, we ran 20 experiments for each benchmark.

A comparison between the learners is drawn in Table 6.3. Our approach needs fewer tests than L*. However, it requires more inputs on several of the PCS case study models. This can be attributed to L* being able to learn these systems without processing any counterexamples. By contrast, L* severely lags behind on the scalable systems benchmarks, which require a series of counterexamples. Our approach also largely defeats TTT on these benchmarks, and even on some of the case study models.

[2]See https://gitlab.science.ru.nl/moerman/Yannakakis

Our approach defeats Tomte on all (admittedly very basic) models. In summary, although the approach appears less effective than TTT, it is still competitive and mostly outmatches Tomte and L*.

Table 6.3: Comparison with other learners

Model	SMT		TTT		L*		Tomte	
	tests	inputs	tests	inputs	tests	inputs	tests	inputs
Biometric Passport	220	1057	220	941	333	1143		
MAESTRO	835	4375	860	4437	1190	4718		
MasterCard	839	4379	996	5260	1190	4718		
PIN	757	3945	911	4769	1190	4718		
SecureCode	313	1485	194	682	798	2758		
Visa	796	4770	750	4094	2040	9015		
PCS_1	629	3530	417	2179	657	2682		
PCS_2	71	279	75	196	252	657		
PCS_3	508	2651	476	2472	576	2196		
PCS_4	559	3024	451	2297	576	2196		
PCS_5	1120	6260	417	1753	1308	5340		
PCS_6	1158	6442	457	1977	1308	5340		
Mealy_FIFOSet(2)	6	27	12	38	14	38		
Mealy_FIFOSet(7)	52	481	71	588	235	2494		
Mealy_FIFOSet(10)	179	2152	163	1822	486	6743		
Mealy_Login(2)	37	214	57	242	57	219		
Mealy_Login(3)	89	644	120	704	240	1720		
IORA_Login(1)	33	152					157	580
IORA_FIFOSet(1)	9	31					21	36.5
IORA_Stack(1)	8.5	33					19	34

A reason to why our approach performs worse than TTT on the case study models may have to do with how hypotheses are constructed. Hypotheses constructed by TTT are completed in terms of their output behavior by running new tests. In contrast, our approach constructs hypotheses solely on the basis of counterexamples. For states whose output behavior has not yet been covered by counterexamples, the solver just produces a *guess* which is likely wrong. This may decrease the efficacy of test algorithms which actively use output behaviors to compute distinguishing sequences (as does the algorithm used in the case study models).

We remark that the seemingly better results we achieved on the scalable systems may be due to their simplistic nature. Having only few inputs, these systems don't benefit as much from the smart exploratory tests a learner may execute.

Although the sample size is small, results seem to indicate that whereas FSM learners are efficient, active register automata learners are yet to reach this level of optimization. These learners often resort to expensive counterexample analysis procedures in order to simplify the counterexample, as in shortening it or isolating the relevant data relations. This simplification is needed in order to minimize the counterexample's subsequent impact on the performance of learning. By contrast, our approach does not need such a procedure. One should note however, that the models our approach can learn lack

Table 6.4: Learning models without resets. Last two columns show results from [171].

States	succ	inputs		time(sec)		inputs	time(sec)
		avg	std	avg	std	avg	avg
1	5	2.0	0.0	0.03	0.0	3	0.01
2	5	6.0	9.55	0.13	0.07	9	0.01
3	5	21.0	6.4	0.43	0.11	18	0.01
4	5	47.0	32.9	0.88	0.21	30	0.01
5	5	48.0	21.48	1.89	0.6	43	0.02
6	5	76.0	53.18	12.95	7.7	57	0.05
7	2	71.5	10.61	25.65	2.59	69	0.13
8	1	288.0	0.0	106.35	0.0	83	0.32

succintness (they are unique valued and non-swapping). Consequently, the number of tests may be adversely affected by the number of registers in a system.

The previous experiment compares our passive learning approach used actively, with active learning approaches. Readers might wonder how our approach and implementation perform relative to similar passive learning algorithms. The preliminary version of our work [192] compares an earlier implementation of our SMT-based approach to DFASAT [107], which implements an efficient SAT-based algorithm for DFAs. Results show that our implementation is competitive. These results are very much in line with those of Neider et al. [162]. Therein, an SMT-based approach for DFAs similar to ours compares favorably when matched against other passive learning approaches.

Our final experiment assesses our extension for learning systems without resets using benchmarks from recent related work [171]. These benchmarks involve learning randomly generated Mealy machines of increasing size with 2 input labels and 2 output labels. These models are connected though they may not be minimal. We adapted our random walk algorithm for setting without resets, using a fixed random length of 3. The solver timeout was set to 10 seconds. Table 6.4 illustrates results. Our extension performs and scales worse than the approach in [171] (which scales up to models of 11 states) and does not provide any guarantees of correctness. However, being able to learn such systems by a simple extension showcases the versatility of an SMT-based approach.

Notes on scalability Scalability is the main weakness of our approach. The IORA and RA encodings scaled up to only a maximum of size 2 for stacks and FIFO sets. By comparison, Tomte can learn FIFO sets of size 30 [6]. The Mealy machine encoding scaled up to a size of 13 for the FIFO set, besting the DFA encoding, which only managed 9. Learning can take several minutes due to the large number of times the solver has to be called. Our implementation calls the solver on every new counterexample, and there can be hundreds of counterexamples in a learning run.

We note some of the measures adopted towards improving scalability while maintaining simplicity of our encodings. These measures largely pertain to how the definitions were implemented into Z3Py. We initially defined sorts for states, locations, labels, inputs and outputs using *Datatype* constructs, which provide a natural representation for expressing sets. We later found that defining a custom sort by using *DeclaredSort* was more efficient (the Mealy machine and DFA FIFO set examples could only scale to 10 and 6 states respectively when using the *Datatype* formulation). This optimization was used for the DFA and Mealy machine encodings. We also found that it was often useful to avoid universal quantifiers (and instead expand them to quantifier-free formulas), in particular when formulating constraints over nodes. Finally, the time the solver took to provide a solution increased with the number of nodes in the OT. This number in turn grows as the tests generated get longer, resulting in longer counterexamples. To give an example of the implication, configuring the test algorithm of [191] to generate longer tests using a random sequence of size 2 instead of 1, meant the VISA model could no longer be learned reliably with a solver timeout of 1 minute.

While some measures where taken to improve scalability, we can definitely see further room for improvement. In particular, the IORA encodings could be made a lot more efficient by utilizing a more succint underlying definition. The current definition requires roughly a doubling of the number of locations, as well as a function to distinguish between input and output locations. A more succint definition would use a transducer-style output function, with each transition encoding both input and output semantics. Another hindering factor is that we still use *Datatype* constructs for implementing both RA and IORA encodings.

6.4 Conclusions

We have experimented with an approach for model learning which uses SMT solvers. The approach is highly versatile, as shown in its adaptations for learning FSMs and register automata, and for learning without resets. We provide an open source-tool implementing these adaptations. Experiments indicate that our approach is competitive with the state-of-the-art. While the approach does not scale well, we have shown that it can be used for learning small models in practice. In the future we wish to improve the scalability of the approach via more efficient encodings. We hope this chapter gives rise to a broader direction of future work, since the presented approach has several advantages over traditional model learning algorithms. Notably, it appears to be quite effective for rapid prototyping of learning algorithms for new formalisms and settings.

Chapter 7

Conclusion

Over the course of this thesis, we have explored two well established model learning approaches for learning Register Automata. In doing so, we have brought each approach closer to its applicability in practice. We have also proposed an alternative approach based on SMT, and shown its effectiveness through a series of experiments. Finally, we have showcased model learning as a viable means of conformance testing. To that end, we have provided compelling applications of learning in the context of two widely used protocols in TCP and SSH. These applications resulted in standard violations, and a notable bug fix of the Linux kernel. Several steps are still needed before learning can become widely applicable. This thesis, nevertheless, paves some steps towards this objective, opening up new opportunities for future research.

7.1 Future Work

State-local automated abstraction refinement One of the key advantages of automated RA learning using mappers as shown in Chapter 4, is its simplicity and flexibility over other approaches. The decoupled architecture of Tomte, the reference tool for this approach, enables us to replace the Mealy machine learner or Lookahead Oracle implementation by any other. This made it possible to easily connect Tomte to the TTT learning algorithm [122] provided by the LearnLib framework [123], resulting in a significant drop in the number of tests. Unfortunately, Tomte's decoupled nature does come at a cost. Experiments revealed not only lack of succintness of the models obtained for larger systems (like the multi-login system with 3 users), but also the exploding number of tests needed to learn them. We attribute this, in part, to reliance on a global set of abstractions, which are encoded in the mapper. This means that in every state, Tomte explores all abstractions instead of only exploring those that are relevant (like the ones that arose in counterexamples).

Isberner et al. [120] have formulated an algorithm that implements the abstractions at the local (state) level. Its application resulted in more succint models and fewer tests needed for learning. The challenge lies in adapting this strategy to the automated abstraction refinement framework of Tomte, while still keeping the learner as decoupled as possible from the mapper. It could be that some integration is required, maybe the learner will be expected to implement some interface, for example, one that would give the suffix and the prefix of the current learning test. Such an interface might still allow easy integration of other FSM learner implementations.

Trees in discrimination trees The tree query approach to learning introduced by Cassel et al. [58] and adopted in Chapter 5, requires a tighter integration between the learner and the tree oracle. The learner has to know about the tree structure, and how different trees are compared, in order to complete its own observation structure and generate a hypothesis out of it. This makes it unlikely that RALib, the reference implementation for this approach, will be able to directly benefit from the extensive framework for FSM learning LearnLib provides. Nevertheless, with some technical effort, advancements from FSM learning can be incorporated into RALib. This is particularly true for those relating to the data structure used to store observations.

RALib builds upon the *observation table* Angluin introduced [19]. While this provides for an intuitive way of storing observations, completing it requires a large number of tests. Many of these tests are not truly needed in order to produce a hypothesis consistent with the last counterexample. Isberner et al. [122] remarked this much, and proposed a much more efficient structure for storing observations, in the form of a *discrimination tree*. This can lead to a marked decrease in the number of tests for learning and overall, as witnessed in Chapter 4. The scalability challenges encountered in the application of RALib to TCP, make a compelling case for adapting RALib so that it uses discrimination trees to store observations.

Passive learners for active learning We have experienced in Chapter 6 the benefits an SMT-based learning approach can provide, namely, how simple it is to develop efficient learners for advanced formalisms by just formulating a few SMT encodings. We also benefited from not having to concern ourselves with counterexamples and their impact on learning performance. On the flip side, we had to face up to the poor scalability of our approach. Less elegant and more efficient encodings would go some way towards improving it. We would have to scale our approach up to machines of moderate size (RAs with 10-30 states), as that's the size protocols typically have.

An alternative would be maintaining the learning framework, but choosing a conventional passive learner instead of an SMT-based one. In other words, perform active learning using a passive learner. That much was done by Walkinshaw et al. [213], but applied to conventional FSMs. Within the same setting, it would be interesting to replace the FSM learner by the MINT tool Walkinshaw et al. developed for learning EFSMs [214, 215]. One may also consider developing new passive learning algorithms for EFSMs, which combine the more scalable SAT-based approaches for FSM inference [106] with the extensive research done on learning EFSMs actively [58].

Leveraging the white box It can sometimes happen that access to the system's code or binary is available. In such cases, techniques such as *dynamic taint analysis* could be leveraged. These could give cheap answers (in terms of tests) to questions such as:

1. What is the message format expected by the machine?
2. What values have been stored on processing the current input?
3. What checks and operations have been applied?

Answering the first question would enable a fully automated learning framework, which also incorporates *interface inference*, not just *model inference.* We mention Autogram [111], a tool which uses taint analysis to extract input formats from Java programs. These formats could then be used in learning.

Answers for the follow-up questions could greatly improve scalability of RA learning techniques. More specifically, in Tomte's framework we could replace the expensive lookahead oracle (the memorable value fetcher) by a simple query to a tainting tool, which would yield the values stored so far without running any tests. Such a tool would also benefit RALib's framework, as knowing the memorable values can significantly reduce the number of tests needed to answer a tree query. Not only that, but it could also make visible to the learner internal operations, that is, operations which don't lead to an immediate output. In particular, it could make visible internal register updates such as the increment of a variable. Determining that such updates took place in a black-box setting can be difficult and may be necessary to learn the system.

Finally leveraging techniques such as *symbolic execution* and *fuzzing* could help us develop more effective test algorithms for learning. The advantage of applying these methods was already shown in previous works [94, 145, 196].

Case studies Learning can be applied to protocols not yet tackled such as QUIC or LDAP. Also, as RA learning algorithms progress some of the previous case studies may be re-visited. In particular, the TCP case study of Chapter 5 involving RALib could only generate models with a limited alphabet due to poor scalability. With future advancements in RALib, a larger alphabet may be used for learning. Case studies where abstract alphabets and a manual mapper were used, such as those on SSH (see Chapter 3) or TLS (see [181]), could be re-done using an RA learner instead of an FSM learner. Some of the parameters abstracted away by the mapper, such as SSH's sequence number or session identifier, could be extracted from the mapper and processed automatically by the learner instead. Once an RA model is learned, we can then verify it against specifications using a model checker adequate for models with data such as nuXmv [166].

Bibliography

[1] *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[2] F. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. PhD thesis, Radboud University Nijmegen, Oct. 2014.

[3] F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop, IEEE International Conference on*, pages 461–468, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[4] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. Vaandrager. Source code and data relevant for the paper 'Learning Register Automata with Fresh Value Generation'. 2017. doi: 10.17026/dans-zkb-4ppm.

[5] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. In *ICTAC 2015*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015.

[6] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. *Unpublished*, 2016.

[7] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *18th International Symposium on Formal Methods (FM 2012), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, Aug. 2012.

[8] F. Aarts, F. Howar, H. Kuppens, and F. Vaandrager. Algorithms for inferring register automata - A comparison of existing approaches. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu,*

Greece, October 8-11, 2014, Proceedings, Part I, volume 8802 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2014.

[9] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *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.

[10] 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*, 46(1):1–41, 2015.

[11] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.

[12] F. Aarts and F. Vaandrager. Learning I/O automata. In *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.

[13] American fuzzer lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2017-08-26.

[14] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, and B. V. Schmidt. *Model-Based Mutation Testing of an Industrial Measurement Device*, pages 1–19. Springer International Publishing, Cham, 2014.

[15] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Softw. Test. Verif. Reliab.*, 25(8):716–748, Dec. 2015.

[16] B. K. Aichernig and M. Tappler. Learning from faults: Mutation testing in active automata learning. In *NASA Formal Methods Symposium*, pages 19–34. Springer, 2017.

[17] aide. http://aide.codeplex.com/. Accessed: 2017-08-27.

[18] M. Albrecht, K. Paterson, and G. Watson. Plaintext recovery attacks against SSH. In *SP'09*, pages 16–26, Washington, DC, USA, 2009. IEEE.

[19] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[20] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5(2):121–150, 1990.

[21] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer*

and Communications Security, CCS '16, pages 1690–1701, New York, NY, USA, 2016. ACM.

[22] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in black: towards formal, black box analysis of sanitizers and filters. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 91–109. IEEE, 2016.

[23] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, Cambridge, Massachusetts, 2008.

[24] M. Büchler, K. Hossen, P. F. Mihancea, M. Minea, R. Groz, and C. Oriat. Model inference and security testing in the spacios project. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 411–414, Feb 2014.

[25] M. Bellare, T. Kohno, and C. Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, 2004.

[26] M. Benedikt, C. Ley, and G. Puppis. Minimal memory automata. In *Alberto Mendelzon Workshop on Foundations of Databases*, 2010.

[27] M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proceedings of AMW 2010*, volume 619, pages http–ceur. CEUR-WS. org, 2010.

[28] J. Berendsen, B. Gebremichael, F. Vaandrager, and M. Zhang. Formal specification and analysis of Zeroconf using Uppaal. *ACM Transactions on Embedded Computing Systems*, 10(3), Apr. 2011.

[29] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE 2005*, pages 175–189. Springer, 2005.

[30] T. Berg, B. Jonsson, and H. Raffelt. *Regular Inference for State Machines with Parameters*, pages 107–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[31] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. *Fundamental Approaches to Software Engineering*, pages 317–331, 2008.

[32] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. pages 535–552, May 2015.

[33] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[34] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, June 1972.

[35] M. Bojańczyk, B. Klin, and S. Lasota. Automata theory in nominal sets. *arXiv preprint arXiv:1402.0897*, 2014.

[36] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In *IJCAI*, volume 9, pages 1004–1009, 2009.

[37] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In *Developments in Language Theory: 17th International Conference, DLT 2013, Marne-la-Vallée, France, June 18-21, 2013. Proceedings*, pages 118–130, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[38] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: The automata learning framework. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2010.

[39] E. Boss. Evaluating implementations of SSH by means of model-based testing. Bachelor's thesis, Radboud University, 2012.

[40] G. Bossert and F. Guihéry. Security evaluation of communication protocols in common criteria. In *Proc of IEEE International Conference on Communications*, 2012.

[41] G. Bossert, F. Guihéry, and G. Hiet. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 51–62, New York, NY, USA, 2014. ACM.

[42] G. Bossert, G. Hiet, and T. Henin. Modelling to simulate botnet command and control protocols for the evaluation of network intrusion detection systems. In *2011 Conference on Network and Information Systems Security*, pages 1–8, May 2011.

[43] M. Botinčan and D. Babić. Sigma*: Symbolic learning of input-output specifications. *SIGPLAN Not.*, 48(1):443–456, Jan. 2013.

[44] R. Braden. *Requirements for Internet Hosts - Communication Layers.* RFC Editor, Fremont, CA, USA, Oct. 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864, 8029.

[45] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, RFC Editor, Fremont, CA, USA, Mar. 1997. Updated by RFC 8174.

[46] E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, January 2004.

[47] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[48] G. Bruns and M. Staskauskas. Applying formal methods to a protocol standard and its implementations. In *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1998),* 20-21 April, 1998, Kyoto, Japan, pages 198–205. IEEE Computer Society, 1998.

[49] M. Bruynooghe, H. Blockeel, B. Bogaerts, B. D. Cat, S. D. Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with idp3. *Theory and Practice of Logic Programming*, 15(6):783–817, 2015.

[50] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, Aug. 1986.

[51] J. Caballero and D. Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451 – 474, 2013. Botnet Activity: Analysis, Detection and Shutdown.

[52] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[53] S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data.* PhD thesis, University of Uppsala, 2015.

[54] S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS 2015*, 2015.

[55] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *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.

[56] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *Journal of Logical and Algebraic Methods in Programming*, 84(1):54–66, 2015.

[57] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. *Inferring Canonical Register Automata*, pages 251–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[58] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, Apr 2016.

[59] G. Chalupar, S. Peherstorfer, E. Poll, and J. d. Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT'14),* San Diego, California, Los Alamitos, CA, USA, Aug. 2014. IEEE Computer Society.

[60] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *NDSS*. The Internet Society, 2004.

[61] D. Chivilikhin and V. Ulyantsev. Muacosm: a new mutation-based ant colony optimization algorithm for learning finite-state machines. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 511–518. ACM, 2013.

[62] D. Chivilikhin, V. Ulyantsev, and A. Shalyto. Combining exact and metaheuristic techniques for learning extended finite-state machines from test scenarios and temporal properties. In *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*, pages 350–355. IEEE, 2014.

[63] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, pages 139–154, 2011.

[64] 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 *ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010.

[65] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[66] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[67] 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.

[68] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[69] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[70] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[71] Conformiq. https://www.conformiq.com/products/. Accessed: 2017-08-18.

[72] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[73] 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.

[74] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM.

[75] L. De Alfaro, T. A. Henzinger, and R. Majumdar. Discounting the future in systems theory. In *ICALP*, volume 2719, pages 1022–1037. Springer, 2003.

[76] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[77] F. H. Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* PhD thesis, Radboud University Nijmegen, July 2012.

[78] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis, 2002.

[79] N. Dodoo, L. Lin, and M. D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical report, 2003.

[80] S. Drews and L. D'Antoni. *Learning Symbolic Automata*, pages 173–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[81] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[82] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[83] O. Esoul and N. Walkinshaw. Using segment-based alignment to extract packet structures from network traces. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 398–409, July 2017.

[84] P. Fiterău-Broştean and F. Howar. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Critical Systems: Formal Methods and Automated Verification*, pages 185–200. Springer, 2017.

[85] P. Fiterău-Broştean and F. Howar. Source code and data relevant for the paper 'Learning-Based Testing the Sliding Window Behavior of TCP Implementations', 2017. doi: 10.17026/dans-zkt-t8xx.

[86] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Source code and data relevant for the paper 'Combining Model Learning and Model Checking to Analyze TCP Implementations'. 2017. doi: 10.17026/dans-xhw-8tyc.

[87] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning Fragments of the TCP Network Protocol. In *FMICS 2014*, volume 8718 of *LNCS*, pages 78–93. Springer, 2014.

[88] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV 2016*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.

[89] P. Fiterău-Broştean, T. Lenaerts, J. de Ruiter, E. Poll, F. Vaandrager, and P. Verleg. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151. ACM, 2017.

[90] P. Fiterău-Broştean, E. Poll, F. Vaandrager, T. Lenaerts, J. D. Ruiter, and P. Verleg. Source code and data relevant for the paper 'Model Learning and Model Checking of SSH Implementations'. 2018. doi: 10.17026/dans-z6n-dxq6.

[91] C. C. Florêncio and S. Verwer. Regular inference as vertex coloring. *Theoretical Computer Science*, 558:18–34, 2014.

[92] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, Jun 1991.

[93] A. Futoransky and E. Kargieman. An attack on CRC-32 integrity checks of encrypted channels using CBC and CFB modes. 1998.

[94] D. Giannakopoulou, F. Howar, M. Isberner, T. Lauderdale, Z. Rakamarić, and V. Raman. Taming test inputs for separation assurance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 373–384, New York, NY, USA, 2014. ACM.

[95] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. *Static Analysis*, pages 248–264, 2012.

[96] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[97] Graphwalker. http://graphwalker.github.io/. Accessed: 2017-08-18.

[98] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. *TCS*, 411(47):4029–4054, 2010.

[99] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.

[100] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[101] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In *FASE 2002*, volume 2306 of *LNCS*, pages 80–95. Springer, 2002.

[102] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[103] Heartbleed bug. http://heartbleed.com/. Accessed: 2017-08-24.

[104] Half a million widely trusted websites vulnerable to heartbleed bug. https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html. Accessed: 2017-08-24.

[105] How to prevent the next heartbleed. https://www.dwheeler.com/essays/heartbleed.html. Accessed: 2017-08-24.

[106] M. Heule and S. Verwer. Exact DFA identification using SAT solvers. In *ICGI*, volume 6339, pages 66–79. Springer, 2010.

[107] M. H. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[108] C. d. Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, Apr. 2010.

[109] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley, 2004.

[110] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Boston, MA, 1979.

[111] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 720–725. ACM, 2016.

[112] K. Hossen, R. Groz, C. Oriat, and J.-L. Richier. Automatic generation of test drivers for model inference of web applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 441–444. IEEE, 2013.

[113] F. Howar. *Active learning of interface programs.* PhD thesis, University of Dortmund, June 2012.

[114] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *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.

[115] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2012.

[116] F. Howar, B. Steffen, and M. Merten. *From ZULU to RERS*, pages 687–704. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[117] 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.

[118] Impacket. http://www.coresecurity.com/corelabs-research/open-source-tools/impacket. Accessed: 2016-01-28.

[119] M. Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective.* PhD thesis, TU Dortmund, 2015.

[120] M. Isberner, F. Howar, and B. Steffen. Inferring automata with state-local alphabet abstractions. In *NASA Formal Methods Symposium*, pages 124–138. Springer, 2013.

[121] M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.

[122] M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 307–322. Springer, 2014.

[123] M. Isberner, F. Howar, and B. Steffen. *The Open-Source LearnLib*, pages 487–495. Springer, 2015.

[124] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.

[125] Junit. https://github.com/junit-team/junit5. Accessed: 2017-08-18.

[126] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[127] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory.* MIT press, 1994.

[128] Keyword-driven testing. https://www.stickyminds.com/article/keyword-driven-testing. Accessed: 2017-08-18.

[129] A. Khalili, L. Natale, and A. Tacchella. *Reverse Engineering of Middleware for Verification of Robot Control Architectures*, pages 315–326. Springer International Publishing, Cham, 2014.

[130] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[131] P. Koopman, P. Achten, and R. Plasmeijer. *Model-Based Shrinking for State-Based Testing*, volume 8322 of *LNCS*, pages 107–124. Springer, 2014.

[132] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[133] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 178–189, New York, NY, USA, 2014. ACM.

[134] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach (6th Edition).* Pearson, 6th edition, 2012.

[135] I. v. Langevelde, J. Romijn, and N. Goga. Founding FireWire bridges through Promela prototyping. In 8^{th} *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA).* IEEE Computer Society Press, April 2003.

[136] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, 1994.

[137] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[138] T. Lenaerts. Improving protocol state fuzzing of SSH. Bachelor's thesis, Radboud University, 2016.

[139] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229, pages 436–450. Springer, 2006.

[140] libalf. http://libalf.informatik.rwth-aachen.de/. Accessed: 2017-08-27.

[141] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15, 2008.

[142] L. Lockefeer, D. M. Williams, and W. J. Fokkink. Formal specification and verification of TCP extended with the window scale option. In *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, volume 8718 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.

[143] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *FMSD*, 6(1):11–44, 1995.

[144] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[145] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. *JDart: A Dynamic Symbolic Analysis Framework*, pages 442–459. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[146] O. Maler and I.-E. Mens. A Generic Algorithm for Learning Symbolic Automata from Membership Queries.

[147] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100. IEEE, 2004.

[148] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126, Nov 2008.

[149] K. Meinke. Automated black-box testing of functional correctness using function approximation. *SIGSOFT Softw. Eng. Notes*, 29(4):143–153, July 2004.

[150] K. Meinke and F. Niu. *A Learning-Based Approach to Unit Testing of Numerical Software*, pages 221–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[151] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In *Tests and Proofs*, pages 134–151. Springer, 2011.

[152] K. Meinke and M. A. Sindhu. Lbtest: A learning-based testing tool for reactive systems. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 447–454, March 2013.

[153] I.-E. Mens and O. Maler. Learning Regular Languages over Large Ordered Alphabets. *Logical Methods in Computer Science*, 11(3), Sept. 2015.

[154] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.

[155] P. F. Mihancea and M. Minea. Jmodex: Model extraction for verifying security properties of web applications. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 450–453, Feb 2014.

[156] J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 613–625, New York, NY, USA, 2017. ACM.

[157] M. Mues, F. Howar, K. Luckow, T. Kahsai, and Z. Rakamarić. Releasing the psyco: Using symbolic search in interface generation for Java. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2017.

[158] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, pages 155–168. USENIX, 2004.

[159] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002.

[160] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[161] D. Neider. Computing minimal separating DFAs and regular invariants using SAT and SMT solvers. In *ATVA*, pages 354–369. Springer, 2012.

[162] D. Neider. *Applications of automata learning in verification and synthesis*. PhD thesis, RWTH Aachen University, Apr. 2014.

[163] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[164] O. Niese. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, 2003.

[165] The nusmv model checker. http://nusmv.fbk.eu/. Accessed: 2017-09-06.

[166] The nuxmv model checker. http://nusmv.fbk.eu/. Accessed: 2017-09-06.

[167] K. Paterson and G. Watson. Plaintext-Dependent decryption: A formal security treatment of SSH-CTR. In *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 345–361. Springer, 2010.

[168] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP Implementation Problems. RFC 2525 (Informational), Mar. 1999.

[169] Pcapy. http://www.coresecurity.com/corelabs-research/open-source-tools/pcapy. Accessed: 2016-01-28.

[170] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.

[171] A. Petrenko, F. Avellaneda, R. Groz, and C. Oriat. From passive to active FSM inference via checking sequence construction. In *IFIP International Conference on Testing Software and Systems*, pages 126–141. Springer, 2017.

[172] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2013.

[173] E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS'07*, pages 164–177, 2007.

[174] E. Poll and A. Schubert. Rigorous specifications of the SSH transport layer. Technical report, Radboud University, 2011.

[175] K. Popper. *Logik der Forschung.* Julius Springer Verlag, Vienna, 1935.

[176] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[177] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.

[178] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

[179] A. Ramaiah, R. Stewart, and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961 (Proposed Standard), Aug. 2010.

[180] Rers 2016. http://rers-challenge.org/2016/. Accessed: 2017-08-26.

[181] J. d. Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security*, pages 193–206, Washington, D.C., 2015. USENIX Association.

[182] H. Sakamoto. Learning simple deterministic finite-memory automata. In *Algorithmic Learning Theory*, pages 416–431. Springer, 1997.

[183] Scapy. http://www.secdev.org/projects/scapy/. Accessed: 2016-01-28.

[184] M. Schuts, J. Hooman, and F. Vaandrager. *Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report*, pages 311–325. Springer International Publishing, Cham, 2016.

[185] Selenium. http://www.seleniumhq.org/. Accessed: 2017-08-18.

[186] M. Shahbaz. Incremental inference of black-box components to support integration testing. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, pages 71–74, Aug 2006.

[187] M. Shahbaz and R. Groz. Inferring Mealy Machines. *FM*, 9:207–222, 2009.

[188] M. Shahbaz and R. Groz. Analysis and testing of black-box component-based systems by inferring partial models. *Softw. Test., Verif. Reliab.*, 24(4):253–288, 2014.

[189] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. *TestCom/FATES*, 4581:319–334, 2007.

[190] W. Smeenk. Applying automata learning to complex industrial software. *Master's thesis, Radboud University Nijmegen*, 2012.

[191] W. Smeenk, J. Moerman, D. Jansen, and F. Vaandrager. Applying automata learning to embedded control software. In *ICFEM 2015*, volume 9407 of *LNCS*, pages 1–17. Springer, 2015.

[192] R. Smetsers. Grammatical Inference as a Satisfiability Modulo Theories Problem. *arXiv preprint arXiv:1705.10639*, 2017.

[193] R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager. Source code and data relevant for the paper 'Model Learning as a Satisfiability Modulo Theories Problem', 2017. doi: 10.17026/dans-xn2-yewe.

[194] R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager. Model Learning as a Satisfiability Modulo Theories Problem. *To appear in LATA 2018.*

[195] R. Smetsers, J. Moerman, and D. N. Jansen. *Minimal Separating Sequences for All Pairs of States*, pages 181–193. Springer International Publishing, Cham, 2016.

[196] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer. Complementing model learning with mutation-based fuzzing. *arXiv preprint arXiv:1611.02429*, 2016.

[197] R. Smetsers, M. Volpato, F. Vaandrager, and S. Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *International Conference on Grammatical Inference*, number 12, pages 167–181. JMLR Workshop and Conference Proceedings 34, 2014.

[198] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.

[199] M. Stoelinga. Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing Journal*, 14(3):328–337, 2003.

[200] M. Tappler, B. K. Aichernig, and R. Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287, March 2017.

[201] Tomte. http://tomte.cs.ru.nl/. Accessed: 2017-09-06.

[202] J. Tretmans. Model-based testing and some steps towards test-based modelling. In *SFM 2011*, LNCS, pages 297–326. Springer, 2011.

[203] O. Udrea, C. Lumezanu, and J. Foster. Rule-based static analysis of network protocol implementations. *Inf. and Comp.*, 206(2-4):130–157, 2008.

[204] V. Ulyantsev and F. Tsarev. Extended finite-state machine induction using SAT-solver. In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*, volume 2, pages 346–349. IEEE, 2011.

[205] F. Vaandrager. Model learning. *CACM*, 60(2):86–95, 2017.

[206] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, Nov. 1984.

[207] P. van den Bos, R. Smetsers, and F. Vaandrager. *Enhancing Automata Learning by Log-Based Metrics*, pages 295–310. Springer International Publishing, Cham, 2016.

[208] P. Verleg. Inferring SSH state machines using protocol state fuzzing. Master's thesis, Radboud University, 2016.

[209] S. Verwer. *Efficient Identification of Timed Automata — Theory and Practice.* PhD thesis, Delft University of Technology, Mar. 2010.

[210] M. Volpato and J. Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 220–235. Springer, 2014.

[211] M. Volpato and J. Tretmans. Approximate active learning of nondeterministic input output transition systems. *ECEASST*, 72, 2015.

[212] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 209–218, Oct 2007.

[213] N. Walkinshaw, J. Derrick, and Q. Guo. *Iterative Refinement of Reverse-Engineered Models by Model-Based Testing*, pages 305–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[214] N. Walkinshaw and M. Hall. Inferring computational state machine models from program executions. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 122–132. IEEE, 2016.

[215] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[216] Y. Wang, X. Li, J. Meng, Y. Zhao, Z. Zhang, and L. Guo. Biprominer: Automatic mining of binary protocol features. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 179–184, Oct 2011.

[217] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, Oct 2012.

[218] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo. Inferring protocol state machine from network traces: a probabilistic approach. In *International Conference on Applied Cryptography and Network Security*, pages 1–18. Springer, 2011.

[219] S. Williams. Analysis of the SSH key exchange protocol. In *Cryptography and Coding*, volume 7089 of *LNCS*, pages 356–374. Springer, 2011.

[220] S. Windmüller, J. Neubauer, B. Steffen, F. Howar, and O. Bauer. Active continuous quality control. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 111–120, New York, NY, USA, 2013. ACM.

[221] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), Jan. 2006.

[222] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), Jan. 2006.

[223] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[224] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), Jan. 2006. Updated by RFC 6668.

[225] H. Yoo and T. Shon. Inferring state machine using hybrid teacher. In *Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on*, pages 504–509. IEEE, 2016.

[226] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 362–372. ACM, 2014.

Curriculum Vitae

Paul Fiterău-Broştean

Born on 09.08.1988 in Timişoara, Romania

2007 - 2011:
Bachelor of Science
Computers and Information Technology
Politechnica University of Timişoara

2011 - 2013:
Master of Science
Software Engineering
Politechnica University of Timişoara

2012 - 2013:
Exchange Student
Radboud University Nijmegen

2013 - 2018:
PhD
Software Science
Radboud University Nijmegen

Titles in the IPA Dissertation Series since 2015

Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning.* Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections.* Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors.* Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).* Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data.* Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities.* Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04