Learning Component Behavior from Tests

Theory and Algorithms for Automata with Data

SOFIA CASSEL
Abstract

Formal models are often used to describe the behavior of a computer program or component. Behavioral models have many different usages, e.g., in model-based techniques for software development and verification, such as model checking and model based testing.

The application of model-based techniques is hampered by the current lack of adequate models for most actual systems, largely due to the significant manual effort typically needed to construct them. To remedy this, automata learning techniques (whereby models can be inferred by performing tests on a component) have been developed for finite automata that capture control flow. However, many usages require models also to capture data flow, i.e., how behavior is affected by data values in method invocations and commands. Unfortunately, techniques are less developed for models that combine control flow with data.

In this thesis, we extend automata learning to infer automata models that capture both control flow and data flow. We base our technique on a corresponding extension of classical automata theory to capture data.

We define a formalism for register automata, a model that extends finite automata by adding registers that can store data values and be used in guards and assignments on transitions. Our formalism is parameterized on a theory, i.e., a set of relations on a data domain. We present a Nerode congruence for the languages that our register automata can recognize, and provide a Myhill-Nerode theorem for constructing canonical register automata, thereby extending classical automata theory to register automata.

We also present a learning algorithm for register automata: the new SL* algorithm, which extends the well-known L* algorithm for finite automata. The SL* algorithm is based on our new Nerode congruence, and uses a novel technique to infer symbolic data constraints on parameters. We evaluated our algorithm in a black-box scenario, inferring, e.g., the connection establishment phase of TCP and a priority queue, in addition to a number of smaller models. The SL* algorithm is implemented in a tool, which allows for use in more realistic settings, e.g., where models have both input and output, and for directly inferring Java classes.

Sofia Cassel, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Sofia Cassel 2015

ISSN 1651-6214
ISBN 978-91-554-9395-0
urn:nbn:se:uu:diva-265369 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-265369)
Nothing endures.
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I  S. Cassel, F. Howar, B. Jonsson, M. Merten, B. Steffen:  
A succinct canonical register automaton model.  
REvised version of:  
A Succinct Canonical Register Automaton Model.  

II  F. Howar, B. Steffen, B. Jonsson, S. Cassel:  
Inferring Canonical Register Automata.  

III  S. Cassel, F. Howar, B. Jonsson, B. Steffen:  
Active Learning for Extended Finite State Machines.  
Technical report 2015-032, Dept. of IT, Uppsala University, October 2015.  
REvised version of:  
Learning Extended Finite State Machines.  

IV  S. Cassel, F. Howar, B. Jonsson:  
RALib: A LearnLib extension for inferring EFSMs.  

Reprints were made with permission from the publishers.
Comments on my participation

Paper I
The ideas originated and were developed in discussions between the authors. I had the main role in writing the paper.

Paper II
I participated in the discussions and writing of the paper.

Paper III
This paper originated from a discussion with a co-author about extending the approach in Paper II, and about integrating the theory with that in [33]. The ideas were developed in discussions between the authors. I had the main role in writing the paper.

Paper IV
The development of ideas and implementation were carried out in collaboration with a co-author. I had the main role in writing the paper.
Other publications

F. Howar, B. Jonsson, M. Merten, S. Cassel:
**On Handling Data in Automata Learning - Considerations from the CONNECT Perspective.**

M. Merten, F. Howar, B. Steffen, S. Cassel, B. Jonsson:
**Demonstrating Learning of Register Automata.**

S. Cassel, B. Jonsson, F. Howar, B. Steffen:
**A Succinct Canonical Register Automaton Model for Data Domains with Binary Relations.**
Sammanfattning på svenska

Denna avhandling handlar om beskrivningar av datorprogram eller komponenter, och om hur vi kan generera sådana beskrivningar automatiskt – utan att egentligen veta hur programmets kod ser ut eller hur komponenten är uppbyggd.

Ponera att vi har en okänd apparat framför oss. Apparaten har olika knappar, men knapparna har ingen text på sig, så vi vet inte vad de kan tänkas ha för funktion. Vår uppgift är att lista ut hur apparaten beter sig i olika situationer. Ett sätt att göra det är att testa den, d.v.s. trycka på olika knappar och observera vad som händer. Om vi testar apparaten tillräckligt utförligt, kan vi sedan göra en beskrivning av dess beteende utifrån våra observationer. Det är detta sätt att generera beskrivningar av program eller komponenter som vi fokuserar på i denna avhandling.

Modeller


Finita automater

I den här avhandlingen är vi intresserade av en speciell typ av modeller, nämligen finita automater. En finit automat består av en ändlig mängd tillstånd som är sammankopplade med övergångar. Figur 1 visar en automatmodell av digitalkameran vi nämnde ovan. Automaten befinner sig alltid i något av de tre tillstånden AV, STANDBY respektive KAMERAN TAR BILDER. Tillstånden beskriver vilka input kameran kan ta emot (här, knapptryckningar). Beroende på vilket input komponenten får, kan den förflytta sig till ett annat tillstånd genom att följa pilarna i figuren.


Forskningsproblem: Finita automater som modeller av komplexa komponenter.

Vi vill använda automater för att modellera mer komplexa komponenter, t.ex. en kamera med begränsad lagringskapacitet som vi bestämmer när vi startar kameran. Om vi använder en vanlig deterministisk finit automat för detta, kommer den att behöva ett tillstånd för varje möjlig lagringskapacitet – om den har kapacitet 5 kommer kameran att kunna ta 5 bilder, osv. Det betyder att vi

Automatisk generering av modeller

Ett stort problem med modeller i allmänhet är att de ofta helt enkelt inte existerar. Det är nämligen både tidskrävande och svårt att göra modeller för hand, i synnerhet om det rör sig om komplexa komponenter eller system med många olika detaljer. Av dessa anledningar blir det ofta så att modeller bara genereras för vissa delar av ett system, eller överhuvudtaget inte alls. Ett annat problem med modeller är att de snabbt kan bli gamla och inkorrekta, t.ex. om komponenten de modellerar har utvecklats eller utökats sedan modellen först genererades.


slut frågar Eleven Läraren om den konstruerade modellen är korrekt eller inte. Läraren svarar då antingen ja, om den är det, eller nej, om det finns något beteende som skiljer sig åt mellan modellen och komponenten. I det fallet kan Eleven använda detta avvikande beteende för att förfinna sin modell.

_Forskningsproblem: Automatisk generering av mer komplexa modeller._

För att kunna använda $SL^*$ i mer realistiska scenarier har vi utvecklat ett verktyg, RALib. RALib innehåller en implementation av $SL^*$ tillsammans med några praktiskt orienterade extrafunktioner. Till exempel kan vi använda RALib för att generera modeller direkt från Java-kod. Vi har jämfört RALib med liknande verktyg, och noterat att RALib ofta kan vara mer effektivt och mångsidigt än andra. Vi beskriver verktyget i kapitel 4.3.
Contents

1 Introduction .................................................................................................................. 16
  1.1 Behavioral models .......................................................................................... 16
  1.2 Generating models ....................................................................................... 18
  1.3 Research challenges ..................................................................................... 20
  1.4 In this thesis .................................................................................................. 22

2 Finite state machines .................................................................................................. 23
  2.1 Finite automata and regular languages ......................................................... 23
  2.2 Constructing a deterministic finite automaton from a regular language ....... 25

3 Extended finite state machines .................................................................................. 28
  3.1 Data languages and register automata ............................................................ 29
  3.2 Constructing a register automaton from a data language ......................... 32

4 Algorithms for automata learning .......................................................................... 38
  4.1 The $L^*$ algorithm: inferring finite automata models ............................... 38
  4.2 The $SL^*$ algorithm: inferring register automata models ....................... 44
  4.3 Automata learning in realistic scenarios ....................................................... 50

5 Summaries of papers ................................................................................................. 53
  I A succinct canonical register automaton model .............................................. 53
  II Inferring Canonical Register Automata ....................................................... 54
  III Active Learning for Extended Finite State Machines ............................ 55
  IV RALib: A LearnLib extension for inferring EFSMs .................................. 56

6 Related work ............................................................................................................. 57
  6.1 Representing languages over infinite alphabets ......................................... 57
  6.2 Learning extended finite state machines with $L^*$ .................................. 60

7 Conclusion(s) ............................................................................................................. 63
  7.1 Future work .................................................................................................... 64

References ...................................................................................................................... 66
Acknowledgements

First and foremost, I would like to thank my advisor Bengt Jonsson, without whom this thesis would not be, and my co-advisor Parosh Abdulla. Thank you for your continuous support over the years!

I would like to thank Bernhard Steffen, Maik Merten, and (in particular) Falk Howar for all our collaborations, the meetings we attended together, and the mutual research visits, starting with the CONNECT project. It has been great working with you!

For many interesting discussions, and for making my life as a PhD student a lot more fun than I expected, I would like to thank (in alphabetical order) Aletta Nylén, Andreas Sandberg, Andreína Francisco, Björn Victor, Carl Leonardsson, Faouzi Atig, Farshid Hassani, Jari Stenman, Johannes Borgström, Johannes Åman Pohjola, Joseph D. Scott, Lisa Kaati, Othmane Rezine, Palle Raabjerg, the Unexpected Sheep and Its Followers, Virginia Grande, Xiaoyue Pan, and Yunyun Zhu\(^1\). An additional huge thank you to those who put up with me during the final months of my PhD, and helped with proof-reading – you know who you are.

Finally, I would like to thank all my friends and family, for supporting me and believing in me. In particular, Carl Hallberg – thanks for everything, I really couldn’t have done this without you. Also, my very special thanks to two very special people, Hugo and Alice. I wouldn’t have done this without you. Thank you for helping me focus on the real world, for thinking that a PhD in computer science means you get to be a doctor who repairs computers, and for thinking that princesses and professors are really the same thing.

---

\(^1\)Yunyun also drew the cover picture. Thank you so much, I love it!
1. Introduction

Suppose we are presented with a large, red box. The box has several buttons, and something that looks like a display. What is it, and what can we use it for? At this point, we cannot know. Perhaps it is a bomb, in which case it might blow up at some point; perhaps it is a coffee machine, in which case it might produce a cup of espresso. We could start looking for an operating manual that would tell us what we can do with the box and what kind of behavior we should expect in response. But if there is no manual, perhaps there is a way to somehow generate a description of the box and its behavior?

One way to figure out how the box behaves is to test it (in a controlled environment, obviously, in case it is a bomb). First, however, we need to find out how to interact with it. In this case, a natural way to interact with the box is to use the buttons that we already noticed. Then, we are ready to test it by pushing different buttons and observing what happens. In order to be able to write a proper description, we also need to carefully write down the tests we perform and what the box does in response. If we choose the tests in a systematic way, we can produce a description of how the box behaves in almost any situation.

This thesis is about generating descriptions of components, such as the red box or a computer program. Such descriptions describe how components (or red boxes) behave in response to certain inputs by you, their user. By examining such a description, we can sometimes discover that a component does something strange or unexpected. The descriptions can also be used for other purposes, for example, to facilitate the process of developing large, complex computer systems.

1.1 Behavioral models

A description of how a component behaves is often called a model. Models can illustrate different aspects or properties of components. At the same time, they abstract from less relevant aspects, omitting details accordingly. For example, a model of a digital camera can show that the camera outputs a picture whenever the camera button is pushed, and otherwise stays idle. It can also show that the order of certain inputs to a component matter: perhaps pushing the camera button only results in a picture if you have previously pushed the on/off button. However, the model might not include all details about the inputs, e.g., for how long a button was pushed.

Using models

Models of components can be used for many different purposes. A very promising use for models is in model-driven engineering. Model-driven engineering
is an approach to software development where models are used in each stage of the development process: to specify the intended behavior of a component, to describe the functionality of a component that has already been implemented, and to check whether the implementation contains unwanted errors or exhibits unexpected behavior. Model-driven engineering can be particularly useful when developing large, complex computer systems. Concretely, it has several benefits: First, using models to specify intended behavior forces engineers to consider the system they are developing at an abstract level, before they actually write the program code. This will often lead to better design decisions. Second, models that specify the intended behavior of some part of a system can sometimes be used to automatically generate program code. This means less manual effort is required, and also that the generated code can be known to be correct by construction, provided that the code generation process does not introduce any errors. Third, models can make it easier to check whether the developed system meets customer requirements.

Models are also used in model-based testing [30, 82], where models are used as a basis for generating test cases, and in model checking [15, 36]. In model checking, a model of a component is automatically checked to see whether it satisfies certain desirable properties. In some sense, these properties help define the intended behavior of the component. There are many model checking tools available, both for components written in particular languages (such as C++ or Java) [16, 29], for particular types of components [12, 13, 48] and for constructing particular kinds of models, e.g., models that take into account real-time parameters or the probability of a system behaving in a certain way [60, 61, 62, 63].

Finite state machine models

Finite state machines are often used to represent the control flow of a component, i.e., the order in which it executes instructions or processes different input from its surroundings. A finite state machine can be drawn as a graph: a finite set of nodes connected by directed edges. The nodes are typically called states, and the edges called transitions. Transitions are labeled with symbols that represent instructions or inputs to the component (and sometimes outputs). At any time, the finite state machine is in exactly one of the states, and by following a transition arrow, it can move to another state. A finite state machine is deterministic if in each state, there is only one possible transition per input.

Figure 1.1 shows a small deterministic finite state machine model of a digital camera. The camera has two buttons: on/off, for turning it on and off, and camera, for taking photos. The camera can be in one of three states: it is either OFF, where photos cannot be taken, in STANDBY, where it is on but waiting for the camera button to be pressed, or in a state where it is TAKING PHOTOS repeatedly whenever the camera button is pressed. Whenever the on/off button or the camera button is pushed, the camera may thus move between states according to the arrows.

We can also describe the control flow of a component in terms of a language. A language is a set of words, i.e., sequences of symbols. In this setting, symbols are inputs to the component, methods invoked on it, or commands sent to it. We can then define, e.g., the language of all words that do not cause the component to return an error message or go into an error state, or the language of all words that let a user log in
Figure 1.1. Small model of the control flow of a digital camera with two functions: it can be turned on/off and photos can be taken using the camera button.

to some service provided by a component. A finite automaton is a finite state machine that recognizes a language, in the sense that it accepts all words in the language and rejects all other words. If a language is regular (there are other kinds of languages, which we do not detail here), this entails that it can be recognized by a (deterministic) finite automaton. When we describe the control flow of a component as a language, then the finite automaton that recognizes that language can be used as a model of the component.

A particular regular language can be represented as many different deterministic finite automata. These automata will have different numbers of states and transitions. For each regular language there is, however, only one minimal (in the number of states) deterministic finite automaton that can recognize it. This means that minimal deterministic finite automata are canonical in the sense that they are the standard automaton representation of the regular language that they recognize. Minimal deterministic finite automata can be obtained from non-minimal deterministic finite automata by means of minimization algorithms, but they can also be constructed directly from a regular language by means of the Nerode equivalence. This equivalence partitions words into a finite set of equivalence classes. The equivalence classes can be used to build a minimal finite automaton where each state represents an equivalence class.

Canonical automata are much easier to compare to each other, since we already know that if they differ somehow, they must represent different languages. Canonical models are important, e.g., for equivalence or refinement checking of models, but also in some forms of dynamic analysis.

1.2 Generating models

A problem with models of components is that they are not always available. Creating models by hand is both difficult and time-consuming, especially for complex components or entire systems where many details need to be taken into account. This means that models are often not created at all, or only for particular parts of a larger system. Another problem is that models may be outdated or inaccurate, if the software has evolved since the model was created. To address these problems, different techniques
for automatically generating models with as little human effort as possible have been
developed.

Broadly speaking, there are two main approaches to the problem of automatically
generating models of components. In \textit{static analysis}, a model is created by analyzing
source code. A prerequisite, in order to use static analysis, is that source code is
available. Static analysis can then exploit the fact that the code provides complete
information about the component’s behavior. This means that a model can be created
without the component ever having to be tested or executed in practice. Static analysis
is used in a variety of different settings, for example, to find security flaws in programs
before they are deployed at a larger scale [64], or to analyze medical devices to ensure
that they will work as intended before they are used by real humans [56].

In \textit{dynamic analysis}, a set of tests is performed on the computer component. The
results are collected, and compiled into a model that describes the component, for
example how it behaves in response to certain input. The advantage of this approach
is that there is no need to access the source code since we are only observing the
behavior from outside. Like static analysis, dynamic analysis is also used in a variety
of different settings. There are tools available that, for example, can find invariants
(i.e., statements that should always hold true) in programs [42], and discover security
vulnerabilities in web applications [69]. Dynamic analysis is often divided into \textit{white-
box} and \textit{black-box} techniques. In white-box techniques, the source code is accessible;
in black-box techniques, there is no access to source code, and the only available part
of the component is its external interface, i.e., what methods are available or what
messages can be sent or received by the component.

\section*{Automata learning}

\textit{Automata learning} is a particular form of dynamic analysis that learns regular lan-
guages, and represents them as finite automata. If we can represent the behavior of a
component as a language, then we can use automata learning to infer a model of that
language. First, a set of tests are performed on the component to be modeled, and
the responses are collected. Then, a deterministic finite automaton is generated that
represents the externally observable behavior of the component (i.e., its language).

For example, consider again our digital camera with two buttons (on/off and cam-
era). Suppose we do not know how to take a photo using the camera, but we want
to find out how it can be done. We can test the camera by pushing the buttons. For
example, we could test the following sequences and observe what happens:

- camera, camera : nothing happens
- camera, on/off : nothing happens
- on/off, camera : the camera takes a photo
- on/off, on/off : the camera turns on, then off
- on/off, camera, on/off : the camera takes a photo, then turns off
- on/off, camera, camera : the camera takes two photos in sequence

By testing the camera in this manner, we can eventually construct a model of the
control flow of the camera. In this case, it could look like the one shown in Figure 1.1.

Two main flavors of automata learning are usually distinguished: \textit{passive} automata
learning and \textit{active} automata learning. In passive automata learning, tests on the com-
ponent we want to model have already been performed, and are provided in the form of traces, e.g., from system logs. One problem with passive automata learning is that it is only as accurate as the set of traces will allow: if there is a particular behavior that the traces do not show, then this behavior will not appear in the constructed model. In active automata learning, however, we choose which tests to perform on the component while we are constructing the model. The choice of which test to perform can thus depend on the results of previous tests. This allows us to create more accurate models, and also to refine existing models by performing additional tests when necessary. It also means we can use fewer tests, since we can choose to only perform tests that will provide useful information.

Automata learning has been used in a variety of different settings (see [39] for an overview), for example to generate models of communication protocols [21, 37], Java programs [67, 84], or embedded systems in cars [79], and in security testing [34, 35, 53, 80]. One common aspect of these studies is that the target systems all expose some interface to the outside world, which means we have an idea of how to interact with them.

Perhaps the most well-known algorithm for automata learning is $L^*$ [14], which infers canonical finite automata based on the Nerode equivalence. This algorithm has been implemented in several tools for automata learning, with the most common one being LearnLib [55, 68, 73].

1.3 Research challenges

If we are modeling the control flow of a component, we can let input symbols represent, e.g., methods that can be invoked on the component. In this way, the finite-state model represents constraints on the order in which different methods or commands can be executed on the component. However, sometimes we want to take into account not only a component’s control flow but also its data flow, i.e., relations between data parameters that are used in methods or commands. To do this, we need a more expressive model than finite automata. One particularly interesting formalism is register automata, which have a finite set of states but are extended with a set of registers that can be used to store data values. Transitions in a register automaton have guards that compare data parameters in input symbols to values stored in registers. A register automaton accepts or rejects a sequence of symbols, i.e., words, based on relations between data values. Depending on which operations and tests we can use in guards and assignments, register automata can recognize different classes of data languages. In this thesis, we capture this dependency by letting register automata be parameterized on a theory, i.e., a set of relations on some data domain.

Challenge I: Extending the register automata formalism.

Many proposed register automata models (e.g., [17, 57]) only allow the theory of equality, which imposes restrictions on what components they can model. In our first research challenge, we want to extend register automata formalisms to the setting where data values can be compared for more relations than equality. This will allow us to model more complex components: for example, a register automaton that compares
data values for inequality (<) could be used to model a component where input values always have to be in a particular order, such as a priority queue data structure.

We have addressed this challenge in Paper III, and describe the results in Chapter 3.

**Challenge II: Constructing canonical register automata.**

Recall that canonicity is an important property for finite automata, e.g., because it allows for easier comparisons between models, and because the $L^*$ algorithm infers canonical finite automata. The latter is particularly interesting in our context. The concept of canonicity has, however, proven hard to carry over to the setting of automata over infinite alphabets (such as, for example, timed automata). This is also true for register automata: when there are formalisms for making canonical register automata models (e.g., in [17]), they tend to impose restrictions on, e.g., in what order they can process data symbols, or on the way that data values can be stored in registers.

In our second research challenge, we are interested in developing a canonical register automata formalism that avoids these restrictions. To do this, we will adapt the Nerode equivalence for finite automata so that it takes into account relations between data values.

We have addressed this challenge in Papers I and III, and describe the results in Section 3.

**Challenge III: Adapting the $L^*$ algorithm to the register automata setting.**

In addition to modeling formalisms, we are also interested in a method for constructing the above models using automata learning, particularly the $L^*$ algorithm. In our third research challenge, we want to extend and generalize it to make it usable for inferring more complex models. The natural choice is to use register automata, since those are the models we typically work with.

Extending the $L^*$ algorithm to the register automaton setting comes with certain problems that need to be solved. For example, we need a new way to organize tests and test results, so that we can accurately capture relations between data parameters. A new way to handle counterexamples also needs to be devised, since the one in $L^*$ does not take into account relations between data values.

We have addressed this challenge in Papers II and III, and describe the results in Section 4.2.

**Challenge IV: Providing a useful tool for inferring register automata models.**

Our final challenge deals with making register automata learning more useful when it comes to inferring realistic components. Typically, components produce output that is non-binary (i.e., not just valid/invalid or OK/NOT OK, but different values of some sort). Register automata, however, only accept or reject words, so they cannot be used directly to model such components. In this challenge, we want to adapt our learning algorithm for register automata to the input/output setting without having to reinvent a new algorithm, and provide a tool that can interface between the learning algorithm and real components, such as Java classes.

We have addressed this challenge in Paper IV, and describe the results in Section 4.3.
1.4 In this thesis

This thesis addresses research challenges in both automata theory (Chapters 2 and 3) and algorithms for automata learning (Chapter 4).

Chapter 2 recapitulates classical automata theory background for the reader: finite automata and regular languages. We introduce the Nerode equivalence and the Myhill-Nerode theorem for regular languages in Section 2.2, and recapitulate how these can be used to construct a deterministic finite automaton from a regular language.

Chapter 3 focuses on Challenges I and II. We discuss register automata and show how they can be used to recognize data languages, i.e., languages where data values can be compared using different relations. We describe our own Nerode equivalence and Myhill-Nerode theorem for register automata in Section 3.2 and show how to use them to construct register automata from a data language.

Chapter 4 describes algorithms for automata learning. Section 4.1 recapitulates the $L^*$ algorithm for inferring deterministic finite automata. The algorithm is described in terms of a series of interactions between a Learner and a Teacher, and related to the Nerode equivalence. We show an example of how to infer a small deterministic finite automaton using the algorithm.

Section 4.2 focuses on Challenge III. We present the $SL^*$ algorithm, which is an extension of the $L^*$ algorithm to the setting where inferred models are register automata. This algorithm is also described as a series of interactions between a Learner and a Teacher, though the information exchanged is different from that in $L^*$. We show an example of how to infer a small register automaton model using our algorithm.

Section 4.3 focuses on Challenge IV. In this section, we present RALib, a tool for adapting the $SL^*$ algorithm for use in more realistic scenarios. We describe the main features of RALib, and how it can be used. We give some examples of models that we can infer using our tool.

Following these chapters, Chapter 5 summarizes each of the included papers. Chapter 6 then provides an overview of related work before we conclude in Chapter 7.
2. Finite state machines

This chapter recapitulates the standard theory of finite automata (see also, e.g., [50, 59]). We introduce the reader to finite state machines and show how they can be used to model computer programs or components. We define the type of language that a finite state machine recognizes, and give a Nerode equivalence and a Myhill-Nerode theorem allowing us to construct minimal finite state machines. We then introduce extended finite state machines, in particular register automata, and describe the languages they recognize. We give a symbolic Nerode equivalence and a Myhill-Nerode theorem for register automata.

A finite state machine consumes a sequence of symbols (e.g., letters, numbers, programming commands, ...) and produces some kind of output. The input symbols come from a finite set, called an alphabet. A finite-state machine can be drawn as a set of states connected by transitions. The transitions are labeled with symbols from the finite state machine’s input alphabet. While processing the symbol that labels a transition, the machine can move from one state to another along the direction of the transition. Initial states are those where the finite state machine can begin processing a sequence of input symbols. Final states are those where the finite state machine will stop and accept, if it has finished processing a word. A state can be initial, final, both, or neither.

Example: A coffee maker

We use a coffee maker to demonstrate how a component can be modeled as a finite state machine. We assume that the coffee maker is very simplistic and has only one button. Pushing the button will make the machine brew coffee, provided it has enough coffee beans and water stored to do so. After having brewed one cup, it must be refilled. We can say that the coffee maker has two input symbols in its alphabet, corresponding to the two things a user can do: refill coffee beans and push the button. Moreover, the coffee maker is always in one of two states: STANDBY (waiting to brew coffee) or NEED REFILL. If the coffee maker is in the STANDBY state, it produces output (coffee) whenever the button is pushed, and then moves to the NEED REFILL state. To get back to the STANDBY state, we must refill beans and water. Figure 2.1 shows a model of the coffee maker. The start arrow corresponds to turning the appliance on, after which it alternates the two states depending on input from the user.

2.1 Finite automata and regular languages

A finite state machine, in general, can produce output that is either some sequence of symbols, or that is binary (e.g., ACCEPT or REJECT). In this chapter, we focus on the
latter type. These are often called finite automata. In a finite automaton, all states are marked either accepting or rejecting. Whenever a finite automaton has finished processing a word, it accepts if it is in a final state, and rejects otherwise. In a deterministic finite automaton, each state has exactly one transition (either to another state or back to itself) for each symbol in the input alphabet. Furthermore, there is only one initial state. Formally, a deterministic finite automaton is defined as follows:

**Definition 1** (Deterministic finite automaton). A deterministic finite automaton is a tuple \( \langle \Sigma, Q, q_0, F, T \rangle \), where
- \( \Sigma \) is a finite input alphabet,
- \( Q \) is a finite set of states,
- \( q_0 \in Q \) is the initial state,
- \( F \subseteq Q \) is the set of final states, and
- \( T \) is a subset of \( Q \times \Sigma \times Q \), such that for each \( q \in Q \), each \( a \in \Sigma \), and some \( q' \in Q \), there is exactly one transition \( \langle q, a, q' \rangle \in T \).

Each transition in a deterministic finite automaton is thus of form \( \langle q, a, q' \rangle \), where \( q \) and \( q' \) are locations in \( Q \), and \( a \) is an input symbol. The automaton moves from \( q \) to \( q' \) on input \( a \) iff the transition \( \langle q, a, q' \rangle \) is in \( T \).

**Regular languages**

A language, in the formal sense, is a set of words; each word is a sequence of symbols from an alphabet. In the remainder of this chapter, we will use the example alphabet \( \{a, b\} \) and write, e.g., \( aa \) for a word consisting of the alphabet symbol \( a \) repeated twice.

A *regular* language is a language that can be defined using a regular expression. Regular expressions describe patterns over words; for example, the regular expression \( abb^* \) defines the set of words that start with one \( a \) and then continue with at least one \( b \). Thus \( abb^* \) is also a regular language. In the remainder of this chapter, we will
refer to this language as $L_{abb^*}$. The words $a$ and $ab$, for example, are words in $L_{abb^*}$ whereas the words $bb$ and $aa$, for example, are not.

We can also relate regular languages to finite automata: a language is regular if and only if it is recognized by a finite automaton. For this reason, we often call finite automata acceptors of regular languages, since they accept all words in a regular language and reject all other words.

![Figure 2.2. A deterministic finite automaton for \{a,b\} that recognizes the language $L_{abb^*}$.](image)

**Example: A deterministic finite automaton**

Figure 2.2 shows a deterministic finite automaton with four states, numbered $q_0$ through $q_3$. State $q_0$ is initial (indicated by an incoming arrow). State $q_2$ is accepting (indicated by a double circle); all other states are rejecting. State $q_3$ is a dead end (or a sink state), since there is no way to reach the accepting state from it. For this reason, we draw transitions leading to the sink state as dotted lines.

The automaton recognizes the language $L_{abb^*}$ over the alphabet \{a,b\}. For example, the word $abbb$ is accepted: the automaton starts in $q_0$, moves to $q_1$ by processing $a$, then moves to $q_2$ by processing $b$, and stays there while processing the last two $b$ symbols. The word $aa$ is rejected: the automaton starts in $q_0$, moves to $q_1$ by processing $a$, and then moves to the sink state $q_3$ by processing the next $a$. □

### 2.2 Constructing a deterministic finite automaton from a regular language

For each regular language, there is one particular deterministic finite automaton that is *minimal*, i.e., it has just enough states to recognize the language. In some sense, the minimal deterministic finite automaton is the most ‘efficient’ among all deterministic finite automata that recognize the regular language, since no other such automaton will have fewer states. The *Myhill-Nerode theorem* [70, 71], due independently to Anil Nerode and James Myhill in the 1950s, provides a way to construct a minimal deterministic finite automaton directly from a regular language. Let us see how this can be done.
Recall that a deterministic finite automaton must, by definition, have a finite set of states. In order to construct a deterministic finite automaton that recognizes a regular language, we need to determine a set of states that is both necessary and sufficient in order to recognize the regular language. If we use too many states (for example, one state for each possible word over the alphabet), then the automaton will be unnecessarily large or even infinite (in which case it would not be a deterministic finite automaton). If on the other hand we use too few states, then the automaton will not be able to recognize the language correctly.

To obtain a finite, minimal number of states from a regular language, we introduce an equivalence relation on words. An equivalence relation partitions a set of elements into a number of equivalence classes. We will define a particular equivalence relation on words, called the Nerode equivalence.

**Definition 2 (Nerode equivalence).** Let $L$ be a language over the alphabet $\Sigma$, and let $u$ and $u'$ be words from $\Sigma^*$. Then, $u$ and $w$ are Nerode-equivalent, denoted $u \equiv_L w$ if for any word $v \in \Sigma^*$, we have that $uv$ is in $L$ if and only if $wv$ is in $L$. $\square$

According to the above definition, any language $L$ induces an equivalence relation on the set of words. Two words are equivalent with respect to $L$ if and only if, when extended by any same suffix, the extended words are either both in the language or both not in the language. Thus, if two words are equivalent according to this definition, it means that they will both always exhibit the same behavior when extended by the same suffix. Conversely, if two words are not equivalent, there is a distinguishing suffix that, when appended to both words, will result in one of the extended words being in the language and the other not being in the language. For example, recall the language $L_{abb^*}$. We have that $ab \equiv_{L_{abb^*}} abb$: both words are in the language, and regardless of what suffix we extend them with, they will always behave the same. On the other hand, we have that $a \not\equiv_{L_{abb^*}} b$: they can be distinguished by, e.g., the suffix $b$, since $ab$ is a word in the language, whereas $bb$ is not.

For each Nerode-equivalence class, there is a residual language. The residual language of a language $L$ with respect to a word $u$, denoted $u^{-1}L$, is the set of all words $v$ such that $uv$ is in $L$. For example, in the language $L_{abb^*}$, the word $a$ has the residual language $a^{-1}L_{abb^*} = bb^*$, and the word $b$ has the residual language $b^{-1}L_{abb^*} = \epsilon$, since no word that starts with $b$ is in $L$. We observe that if two words $u$ and $w$ have equal residual languages, i.e., if $u^{-1}L = w^{-1}L$, then there can be no distinguishing suffix between them, i.e., they must be Nerode-equivalent with respect to $L$. Thus $u^{-1}L = w^{-1}L$ implies $u \equiv_L w$.

In a minimal deterministic finite automaton, two words with identical residual languages will always lead to the same state, since there is no distinguishing suffix between them, i.e., they will always behave the same when extended by the same suffix. Moreover, two words $u$ and $w$ whose residual languages differ will always lead to distinct states. When this is the case, there is at least one suffix $v$ for which $uv \in L$ but $wv \not\in L$ (or vice versa). The suffix $v$ is then a distinguishing suffix for $u$ and $w$ with respect to $L$, so both words cannot lead to the same state.

Provided that the Nerode equivalence has finitely many equivalence classes, we can use the Myhill-Nerode theorem to build a deterministic finite automaton where each state directly corresponds to a Nerode-equivalence class. The automaton is guaranteed
by the theorem to be minimal. Conversely, if the Nerode equivalence does not have finitely many classes, then it is not possible to build a deterministic finite automaton that recognizes the language. This, in turn, means that the language is not regular.

**Theorem 1** (Myhill-Nerode). Let \( L \) be a regular language over an alphabet \( \Sigma \), and \( \equiv_L \) the induced Nerode equivalence on \( \Sigma^* \) with respect to \( L \).

- The language \( L \) is regular if and only if the number of equivalence classes of \( \equiv_L \) is finite.
- The minimal deterministic finite automaton that recognizes \( L \) has exactly as many states as the number of equivalence classes of \( \equiv_L \). \( \square \)

For example, consider again the language \( L_{abb^*} \). The equivalence classes of \( \equiv_{L_{abb^*}} \) are four, each defined by a regular expression:

- \( \epsilon \) (the singleton class),
- \( a \) (the singleton class),
- \( abb^* \) (the equivalence class of all words that begin with \( a \) and continue with at least one \( b \)), and
- \( (aa|b)(a|b)^* \) (the equivalence class of all words that begin with \( aa \) or with \( b \)).

It can easily be seen that each of the four classes corresponds to a state in the automaton in Figure 2.2.

The Myhill-Nerode theorem can be used to show that languages are regular (or not), but more importantly for this thesis, it can be used as a basis for automata learning algorithms (Chapter 4).
3. Extended finite state machines

More complex components cannot always be adequately and succinctly represented by finite state machines: if a component’s input alphabet is very large, a finite state machine model of it will become infeasibly large, and if the input alphabet is infinite, the component cannot be represented as a finite state machine at all.

To address this problem, we can instead use extended finite state machines. Extended finite state machines focus on relations between input symbols rather than on their concrete values. They are finite state machines extended with memory: they have assignments, and they have guards over input symbols.

Example: A coffee maker with memory

Suppose we have another coffee maker, more advanced than the one in Chapter 2. The new coffee maker can brew several cups of coffee at a time, so in addition to the one button for making coffee, it also has a keypad where users can enter the desired number of cups. The advanced coffee maker stores coffee beans and water for multiple cups, and when processing user input, it takes into consideration how much coffee beans and water it is currently storing. If the number of cups input by the user is more than the current amount of water and coffee beans will allow, the machine will ask for a refill; otherwise, it will brew coffee. We assume that the coffee maker can store a maximum of ten cups’ worth of water and coffee beans.

The advanced coffee maker cannot be modeled as a finite automaton, since there are infinitely many combinations of digits that can be entered via the keypad, and each of these combinations would need to be represented as a separate transition. To model this coffee maker using a finite number of states requires a more advanced formalism that has memory in addition to states and transitions.

Figure 3.1 shows a model of the advanced coffee maker. It is always in one of two states: STANDBY and NEED REFILL. It stays in the STANDBY state while brewing coffee, until its store of water and coffee beans has been depleted; then, it moves to the NEED REFILL state. To get back to the STANDBY state, we must refill it with some amount of beans and water.

The start arrow corresponds to turning the coffee maker on, and setting the initial quantity of coffee beans and water. Following this, the coffee maker alternates the two states depending on what we do to it. In order for the appliance to function properly, it must always remember how much water and coffee beans it has stored, and be able to update this information as coffee is brewed and water and coffee beans refilled. The coffee maker’s memory is represented as a gray square in the figure, and it is updated by every transition.

□
Figure 3.1. Small model of a coffee maker with memory

3.1 Data languages and register automata

In this thesis, we deal with a particular kind of extended finite state machines, called register automata. Before we discuss register automata, let us describe how we can represent relations between alphabet symbols in terms of a formal language.

A data symbol is of the form \( a(d) \) where \( a \) is an action from a finite set \( \mathcal{A} \) and \( d \) is a data value\(^1\) from a possibly unbounded domain \( D \). We use the term data word to denote a sequence of data symbols. In order to represent relations between data values in a data word, we introduce a theory. A theory is a pair \( \langle D, \mathcal{R} \rangle \), i.e., a set \( \mathcal{R} \) of relations on a data domain \( D \). The theory specifies what types of data values (e.g., integers or real numbers), and what relations between data values (e.g., equality or inequality) are used to determine membership in a particular data language. For example, if the set of relations in a theory is empty, then membership in a data language is only based on the actions in data symbols, similar to a deterministic finite automaton. On the other hand, if the set of relations is equality, then membership is based both on the actions and on whether data values are equal or not. We formalize this as follows:

**Definition 3 (\( \mathcal{R} \)-indistinguishable data words).** Two data words \( \alpha_1(d_1) \ldots \alpha_n(d_n) \) and \( \alpha'_1(d'_1) \ldots \alpha'_n(d'_n) \) are \( \mathcal{R} \)-indistinguishable if

- \( \alpha_1 \ldots \alpha_n = \alpha'_1 \ldots \alpha'_n \), i.e., they have the same sequences of actions, and
- \( R(d_i, \ldots, d_j) \) if and only if \( R(d'_i, \ldots, d'_j) \) whenever \( R \in \mathcal{R} \) and \( i, \ldots, j \) are indices between 1 and \( n \).

\(^1\)In this thesis, we assume that a data symbol only has one data value, but our results extend to the case where data symbols carry more than one data value.
For example, by using the theory of equality over integers, we can distinguish between the words \( a(1)a(4)b(1) \) and \( a(3)a(3)b(3) \) where the sequences of actions are the same, but where, in the one case, only the first and third data values are equal, but in the other case all data values are equal. However, the words \( a(1)a(4)b(1) \) and \( a(5)a(2)b(5) \) are indistinguishable with respect to the equality relation since, in both cases, the first and third data values are equal. If instead we use the theory of equality and inequality \((<)\) over integers, we can also distinguish between the words \( a(1)a(4)b(1) \) and \( a(5)a(2)b(5) \): in the one case, the second data value is bigger than the first and the third data values, but in the other case, the second data value is smaller than the first and the third data values.

A data language is a set of words that respects the set \( R \) of relations, in the sense that whenever two data words are \( R \)-indistinguishable, they are either both in the data language or both not in the data language. If a data language respects the set \( R \) of relations on a data domain \( D \), we say that it is parameterized on the theory \( \langle D,R \rangle \).

Register automata

We are now ready to introduce register automata. We will use them as acceptors for data languages, and, in this thesis, limit ourselves to deterministic register automata. We refer to the class of languages that are recognizable by a deterministic register automaton as regular data languages. Formally, a deterministic register automaton can be defined as follows:

**Definition 4** (Register automaton). A register automaton is a tuple \( \langle L,l_0,F,X,\Gamma \rangle \), where

- \( L \) is a finite set of locations,
- \( l_0 \) is the initial location,
- \( F \subseteq L \) is the set of final (accepting) locations,
- each location \( l \) has a finite set \( X(l) \) of registers, and
- \( \Gamma \) is a finite set of transitions, each of form \( \langle l,\alpha(p),g,\pi,l' \rangle \) where
  - \( l,l' \in L \) are the source and target locations, respectively,
  - \( \alpha(p) \) is a parameterized input symbol with the parameter \( p \),
  - \( g \) is a guard over \( p \) and \( X(l) \), and
  - \( \pi \) is the assignment, a mapping from \( X(l') \) to \( X(l) \cup \{p\} \). Intuitively, the register \( x' \in X(l') \) is assigned the value of \( \pi(x') \), which is either the parameter \( p \) or some register \( x \in X(l) \). \( \square \)

**Example: A register automaton**

A register automaton can be represented as a set of locations with transitions between them. Locations have registers that can store data values. Figure 3.2 shows a register automaton with four locations, numbered \( l_0 \) through \( l_3 \). \( l_0 \) is the initial location (indicated by an arrow); all locations except \( l_3 \) are accepting (indicated by a double circle). Location \( l_3 \) is a sink, meaning that it is rejecting once reached, there is no way to get to an accepting location. All locations except the initial location have a register \( x \).
Transitions are labeled with parameterized symbols and guards over parameters and registers. Transitions that lead to the sink location $l_3$ are denoted by dotted lines. Parameters are placeholders for data values, so parameterized input symbols represent data symbols. Guards represent conditions on data values. For example, the guard on the transition that loops in $l_1$ states that the data value occurring in an $a$-symbol must be at least as big as the data value stored in the register $x$.

The register automaton processes data symbols with actions from the set $\mathbb{A} = \{a\}$ and data values from the domain of natural numbers. Thus, it accepts words of form $a(d_1)\ldots a(d_n)$ under certain conditions. More precisely, the data language that the automaton recognizes consists of words of form $a(d_1)\ldots a(d_n)$, such that whenever $d_i > d_{i+1}$ for some $i \leq (n - 2)$, then $d_{i+1} \leq d_{i+2}$. In the remainder of this chapter, we will refer to this data language as $L_{ra}$.

When the register automaton processes a data symbol, it matches the data symbol with a parameterized input symbol, checks which transition guard is satisfied by the symbol’s data value, and moves along that transition. Let us now see how the register automaton processes a data word.

On the first transition from $l_0$ to $l_1$, the data value occurring in an $a$-symbol is stored in $x$. The register is then updated on transitions to and from location $l_2$. The loop transition in location $l_1$ is taken whenever the data value occurring in an $a$-symbol is at least as big as the one currently stored in $x$. For example, the word $a(1)a(4)a(0)a(7)$ is accepted: the automaton starts in $l_0$ where it stores 1 in the register $x$. Then it loops in location $l_1$ and stores 4, after which it moves to $l_2$ by comparing 0 to the value stored in $x$ (which is 4). It also updates $x$ to 0. Finally, the automaton moves back to $l_1$ since 7 is bigger than the currently stored value 0. □

![Figure 3.2. A register automaton that recognizes $L_{ra}$, i.e., accepts words of the form $a(d_1)a(d_2)\ldots$ under certain conditions.](image)

The model in Figure 3.2 corresponds to the register automata presented in Paper III. In Paper I, we used slightly different terminology\(^2\) to present a register automaton model for the theory of equality over different data domains. There are other similar register automata models (e.g., [17, 57]) that also restrict possible relations between

\(^2\)Paper I does not use the notion of a theory (since only equality is considered) and describes data words in terms of constraints on their data values.
data values to only equality, and otherwise differ slightly, e.g., in whether registers are initially empty or not, and in what values are stored in registers during a run of the automaton.

3.2 Constructing a register automaton from a data language

Recall that a register automaton accepts or rejects words in a data language based on relations between data values. In some sense, we can say that it processes words at a *symbolic* level. In Section 2.2, we described a way to construct a finite automaton from a regular language by means of the Myhill-Nerode theorem. Now, we would like to use a similar approach to construct register automata from a data language. More precisely, we want to define an equivalence relation that classifies prefixes based on whether the result of concatenating them with a particular suffix is in the data language or not. This will allow us to construct locations in a register automaton, but we also need to be able to determine which transition guards to use, and when and how to store data values in registers. Unfortunately, the Nerode equivalence in Definition 2 will not work directly for this; we need to adapt it to the register automata setting, as the following example shows.

Consider the data language $L_{ra}$ accepted by the register automaton in Figure 3.2. We choose two data words $u$ and $w$ that lead to the same location in the register automaton, and see if we can find a distinguishing suffix. Since the Nerode equivalence is used to construct minimal automata, we should not be able to find such a distinguishing suffix if the equivalence is directly adaptable to the register automaton setting.

For example, let $u = a(5)a(2)$ and $u' = a(8)a(4)$. According to Figure 3.2, both these words lead to location $l_2$, with $x$ storing 2 and 4, respectively. However, we can easily find a suffix $v$ that distinguishes them, for example, $v = a(3)$. In the automaton, $uv = a(5)a(2)a(3)$ leads to location $l_1$ whereas $u'v = a(8)a(4)a(3)$ leads to the sink location. In fact, by applying an equivalence relation similar to the Nerode equivalence for finite automata, almost all words could become inequivalent. The reason for this is that by comparing two data words with respect to an individual suffix, we cannot capture relations between data values: in the one case, 3 is smaller than the value currently stored in $x$; in the other case, 3 is bigger. Since register automata accept or reject words based on relations between data values, we need to use a different notion of ‘suffix’ for comparing two words, and amend the Nerode equivalence accordingly. Instead of one suffix, we can instead consider a parameterized suffix (e.g., of form $\alpha_1(p_1) \ldots \alpha_n(p_n)$) and model how relations between the parameters $p_1$, $p_2$ and the data values in the prefix affect whether the resulting data word is in the data language or not. In the remainder of this chapter, we show how this can be done.

Symbolic decision trees

We will define a symbolic version of a suffix, essentially consisting of several suffixes together, represented as different relations between data values. We call these *symbolic*
decision trees. Symbolic decision trees are used instead of suffixes in order to adapt Definition 2 to the symbolic setting, i.e., with register automata and data languages.

A symbolic decision tree is essentially a restricted form of a register automaton in the shape of a tree. It has locations with registers, and transitions with guards and assignments. A \((u,V)\)-tree is a symbolic decision tree that models the acceptance of a specific set of suffixes after a given prefix \(u\). We refer to the sequence of actions in a data word \(w\) as \(Acts(w)\), and use the term symbolic suffix to denote \(Acts(v)\) whenever \(v\) is a suffix. Formally, we define a \((u,V)\)-tree as follows:

**Definition 5 \(((u,V)\)-tree).** Let \(u\) be a data word and let \(V\) be a set of symbolic suffixes. A \((u,V)\)-tree is a symbolic decision tree \(T\), which, for any suffix \(v\) with \(Acts(v) \in V\), either accepts \(uv\) or rejects \(uv\). □

![Figure 3.3](image)

Figure 3.3. A symbolic decision tree that accepts \(a(d)\) for any \(d\).

Figure 3.3 shows a \((u,V)\)-tree with two locations: a root location at the top and a leaf location at the bottom. Both locations are accepting (denoted by double circles). The tree starts processing input in the root location, again similar to a register automaton. It accepts suffixes of form \(a(d)\), regardless of the value of \(d\), e.g., \(a(2)\) or \(a(1)\).

![Figure 3.4](image)

Figure 3.4. Symbolic decision trees \(((u,V)\)-trees) for \(V = \{a\}\) and different \(u\).

The tree in Figure 3.4(a) is slightly more complex, with three locations and one register in the root location, \(x_2\). It accepts suffixes of form \(a(d)\) whenever \(d\) is at least as big as the value stored in the register \(x_2\), and otherwise rejects.

Whenever a \((u,V)\)-tree has registers in the root location, these registers store values from the prefix \(u\). This enables the tree to model relations between data values in \(u\) and data parameters in the set of suffixes, by means of guards and assignments. We introduce a technical restriction on registers, namely that a register \(x_i\) may only store the \(i\)th data value in a word (either from the prefix or suffix). This makes it easier to compare \((u,V)\)-trees to each other.

We use \(\mu_u\) to denote the valuation of registers that a particular prefix \(u\) induces. For example, a \((u,V)\)-tree for \(u = a(8)a(4)\) and \(V = \{a\}\) will look exactly like the tree in
Figure 3.4(a). In this case, \( \mu_u \) specifies that the value of \( x_2 \) is 4 (since \( x_2 \) stores the second data value from \( u \)).

Two \((u,V)\)-trees \( T \) and \( T' \) for the same set \( V \) of symbolic suffixes are isomorphic, denoted \( T \equiv T' \) if their locations, transitions, and registers are the same. For a bijection \( \gamma \) between the registers of \( T \) and the registers of \( T' \), the trees \( T \) and \( T' \) are isomorphic under \( \gamma \), denoted \( T \equiv_{\gamma} T' \) if they are isomorphic when renaming all registers according to \( \gamma \). For example, the trees in Figure 3.4(a) and 3.4(b) are isomorphic under \( \gamma \) whenever \( \gamma(x_2) = x_3 \).

\[ x_1 \]
\[ a(p) \mid p < x_1 \]
\[ x_2 := p \]
\[ a(p) \mid x_1 \leq p \]
\[ a(p) \mid x_2 \leq p \]
\[ a(p) \mid p < x_2 \]
\[ a(p) \]

Figure 3.5. A symbolic decision tree that accepts or rejects \( a(d)a(d) \) under certain conditions.

Symbolic decision trees can be refined by adding more symbolic suffixes to \( V \). For example, the \((u,V)\)-tree in Figure 3.5 refines the tree in Figure 3.3: A register \( x_1 \) has been added, since in the refined tree, guards refer to data values in the prefix. The tree has also increased in length, since in the refined tree we consider suffixes of length 2.

The initial branch from the root location has been split in two, where the left branch represents the case when the data value in \( a(d_1) \) is smaller than the one stored in \( x_1 \) and the right branch represents the case when the data value in \( a(d_1) \) is at least as big as the stored value. The refined tree as a whole distinguishes three different cases, represented by the three leaf nodes:

- The **middle** leaf represents the case where the prefix \( a(2) \) is followed by consecutively smaller \( d_1 \) and \( d_2 \). For example, the word \( a(2)a(1)a(0) \) would reach this leaf.
- The **left** leaf represents the case where the prefix \( a(2) \) is followed by a smaller \( d_1 \) and then a bigger \( d_2 \). For example, the word \( a(2)a(0)a(1) \) would reach this leaf.
- The **right** leaf represents the case where the prefix \( a(2) \) is followed by a bigger or equal \( d_1 \) and then any \( d_2 \). For example, the word \( a(2)a(3)a(2) \) would reach this leaf.

**Tree oracles**

We already have the notion of a symbolic decision tree for a set of symbolic suffixes after a given prefix. Next, we need to ensure that the tree conforms to a particular data language, i.e., that whenever a word \( uv \) is in the data language, a \((u,V)\) tree will accept \( v \) when starting in the initial valuation \( \mu_u \) if \( v \) is in \( V \). We use a function called a **tree oracle** to do this. A tree oracle is formally defined as follows:
We present the following example. The same or extended.

Initial transitions either stay the same or are refined; the set of registers is either the equivalent. Third, whenever we extend the set of symbolic suffixes, the guards of the symbolic suffixes we cannot make two previously inequivalent symbolic decision trees calculated. First, the oracle constructs trees recursively. Second, by extending the set of symbolic suffixes, returns a \((u,V)\)-tree \(\mathcal{O}(u,V)\). This tree has the property that for any data word \(uv\) with \(\text{Acts}(v) \in V\), \(v\) is accepted by \(\mathcal{O}(u,V)\) under \(\mu_u\) if and only if \(uv \in \mathcal{L}\), and rejected by \(\mathcal{O}(u,V)\) under \(\mu_u\) if and only if \(uv \notin \mathcal{L}\).

The tree oracle generates a \((u,V)\)-tree that conforms to a particular language \(\mathcal{L}\). However, Definition 6 in principle does not specify exactly what a tree should look like for a given \(u\), \(V\), and \(\mathcal{L}\). This can be problematic when we want to use the trees to construct a canonical register automaton. In particular, we want trees to always look the same when constructed for the same \(u\), \(V\), and \(\mathcal{L}\), and we want to impose some restrictions on which guards and assignments are allowed. One prerequisite for this is the existence of a representative data value for each guard in a tree, which allows us to generate exactly one concrete data word for each sequence of guarded transitions in a \((u,V)\)-tree. We define a representative data value as follows:

**Definition 7** (Representative data value). For a guard \(g\) and a prefix \(u\), a representative data value \(d^u_g\) is a data value with the following properties:

- it satisfies \(g\) under the valuation \(\mu_u\) of registers induced by \(u\), and
- whenever \(g\) is refined to become \(h\), then \(d^u_h = d^u_g\) provided that \(d^u_g\) satisfies \(h\) under \(\mu_u\).

Now that we have representative data values, we are ready to define canonical tree oracles.

**Definition 8** (Canonical tree oracle). The tree oracle \(\mathcal{O}(\mathcal{L})\) is canonical if it satisfies the following conditions for any data language \(\mathcal{L}\), prefix \(u\), and set \(V\) of symbolic suffixes:

- Let \(\mathcal{O}(u,V)[l]\) denote the subtree of \(\mathcal{O}(u,V)\) starting in location \(l\). For each initial transition \(\langle i_0, \alpha(p), g, \pi, l \rangle\) in \(\mathcal{O}(u,V)\), we have that \(\mathcal{O}(u,V)[l] \equiv \mathcal{O}(u \alpha(d^u_g), \tilde{V})\) where \(\tilde{V}\) contains all symbolic suffixes in \(V\) for which \(\alpha v\) is in \(V\).
- Whenever \(V \subseteq V'\), then
  - for any \(u\), \(u'\) and renaming \(\gamma\), if \(\mathcal{O}(u,V') \equiv_\gamma \mathcal{O}(u',V')\) then \(\mathcal{O}(u,V) \equiv_\gamma \mathcal{O}(u',V)\),
  - for each initial \(\alpha\)-guard \(g\) in \(\mathcal{O}(u,V)\) there is an initial \(\alpha\)-guard \(h\) in \(\mathcal{O}(u,V')\) with \(\mu_u \models h[d^u_g/p]\) and \(\mu_u \models h \rightarrow g\), and
  - the set of registers in the tree \(\mathcal{O}(u,V)\) is a subset of the registers in \(\mathcal{O}(u,V')\).

The definition states the properties that are necessary for a tree oracle to be canonical. First, the oracle constructs trees recursively. Second, by extending the set of symbolic suffixes we cannot make two previously inequivalent symbolic decision trees equivalent. Third, whenever we extend the set of symbolic suffixes, the guards of the initial transitions either stay the same or are refined; the set of registers is either the same or extended.

To give an idea of how symbolic decision trees can be generated by a tree oracle, we present the following example.
Example: *Generating symbolic decision trees*

Consider the two words $u = a(5)a(2)$ and $w = a(8)a(4)$, and the language $L_{ra}$ over the theory of equality and inequality over natural numbers. The set $\mathcal{R}$ of relations is thus $\{=, <\}$. The words $u$ and $w$ are $\mathcal{R}$-indistinguishable. We can compare them at a symbolic level, i.e., focusing on the relations between data values, by doing the following:

- We extend each word with a set of carefully chosen suffixes. By ‘carefully chosen’, we mean one suffix from each class of $\mathcal{R}$-indistinguishable suffixes. Thus, there are five possible suffixes with which each word will need to be extended: after $a(8)a(4)$, for example, these can be $a(3)$, $a(4)$, $a(6)$, $a(8)$, and $a(9)$. Each of these suffixes represents a unique relation between the data value in the suffix and the data values in the prefix. For example, $a(6)$ represents the case where the data value in the suffix is smaller than the first data value in $a(8)a(4)$, but bigger than the second one. (In this case, we might of course just as well have chosen $a(5)$, since $a(8)a(4)a(5)$ is $\mathcal{R}$-indistinguishable from $a(8)a(4)a(6)$.)
- We group the suffixes according to whether the corresponding extended words are in $L_{ra}$ or not. For example, after $a(8)a(4)$, the suffix $a(3)$ is rejected, whereas $a(4)$, $a(6)$, $a(8)$ and $a(9)$ are all accepted.
- We organize the groups of suffixes in a tree structure, and label each branch with the appropriate relation(s) between data parameters. Registers in the root location store data values from the original word. For example, after the word $a(8)a(4)$, we can use a register $x_2$ to denote the stored value 4. The accepted suffixes $a(4)$, $a(6)$, $a(8)$ and $a(9)$ after $a(8)a(4)$ can be collectively represented as $x_2 \leq p$, where $p$ represents the data value occurring in the symbol $a(d)$. The remaining rejected suffix can be represented as $p < x_2$. We leave it as an exercise to the reader to verify that the same holds for the set of five suffixes after $a(5)b(2)$, i.e., that they can be represented using the same tree, modulo the value stored in $x_2$.

When determining the guards with which to label branches in a tree, we use relations that are as weak as possible. This has the effect that we will only store data values from the prefix in registers if they are necessary for determining acceptance or rejection of a suffix. In this case, for example, we do not need any relation between the first data value in the prefix and data parameters in the suffix, since it does not matter for acceptance or rejection of the suffixes.

We are now ready to define a symbolic version of the Nerode equivalence using symbolic decision trees.

**Definition 9** (Symbolic Nerode equivalence). Let $O$ be a canonical tree oracle for a theory $(\mathcal{D}, \mathcal{R})$. Let $u$ and $u'$ be data words. For a language $L$, the words $u$ and $u'$ are Nerode-equivalent if for any set $V$ of symbolic suffixes, there is a bijection $\gamma$ between the registers in $O_L(u, V)$ and $O_L(u', V)$ such that $O_L(u, V) \equiv_{\gamma} O_L(u', V)$.

---

3 This is a simplified version of the equivalence defined in Paper III, which takes into account particular properties of theories.
The symbolic Nerode equivalence states that two data words are equivalent with respect to a data language if and only if their symbolic decision trees are isomorphic under some bijection between registers when constructed from the same set of symbolic suffixes. In other words, if two data words $u$ and $w$ are equivalent according to this definition, it means that whenever a word $v$ is accepted (rejected) by the symbolic decision tree after $u$, then any word $z$ that is $R$-indistinguishable from $v$ is accepted (rejected) by the symbolic decision tree after $w$ (and vice versa).

Conversely, if two data words are not equivalent, there is a set $V$ of symbolic suffixes for which the constructed symbolic decision trees cannot be made isomorphic under any renaming of registers. Much in the same way as the original Nerode equivalence, the symbolic version can thus be used to separate locations when building a register automaton. Provided that the symbolic Nerode equivalence has finitely many equivalence classes, we can use a symbolic version of the Myhill-Nerode theorem to build a register automaton where each state directly corresponds to (at least) one symbolic Nerode-equivalence class. This register automaton will not necessarily be minimal, but canonical. Conversely, if the symbolic Nerode equivalence does not have finitely many equivalence classes, then we cannot build a register automaton that recognizes the data language.

**Theorem 2** (Symbolic Myhill-Nerode). Let $L$ be a data language, and $\equiv_L$ a symbolic Nerode equivalence on $L$. If $L$ is regular$^4$, then we can build a register automaton that recognizes $L$. This automaton will have at most as many locations as the number of equivalence classes of $\equiv_L$. □

We use the symbolic Myhill-Nerode theorem as a basis for the symbolic automata learning algorithm presented in Section 4.2. A proof for this theorem can be found in Paper III.

---

$^4$This is with respect to a particular tree oracle, not necessarily in general. See Paper III for details.
4. Algorithms for automata learning

Finite automata models of (regular) languages can be inferred using a form of machine learning called concept learning. A concept in this sense is a set of objects from some domain. In concept learning, or learning from examples, the task is to infer a model that represents the unknown concept. In order to do this, we need a set of examples from the domain. This set is called training data. Each example in the training data has a label that states whether it belongs to the concept we are learning or not.

There are two main approaches to concept learning, differing in the principles for labeling training data. In passive learning, all training data has been labeled beforehand and no additional training data is available. In active learning, training data is only labeled when the particular learning algorithm chooses to do so. In this thesis, we focus on active concept learning.

Active learning in general, and active concept learning in particular, is often formulated as an interaction between a Learner and a Teacher. The Teacher has knowledge about the unknown concept. The Learner can make queries to the Teacher about the concept, and the Teacher answers these correctly. The Learner uses this information to construct a representation of the concept.

In a minimally adequate teacher (MAT) setting [14], the Teacher can answer two kinds of queries: membership queries and equivalence queries. A membership query consists in asking for the label of a particular example from the domain. An equivalence query is made when the Learner has constructed a hypothesis of the concept, and consists in asking whether this hypothesis correctly represents the concept or not. If not, the Teacher supplies a counterexample: an example that is not part of the concept but in the Learner’s hypothesis, or vice versa. The efficiency of algorithms that work in the MAT setting is typically measured in terms of the number of membership queries and equivalence queries needed infer a representation of a particular concept.

4.1 The $L^*$ algorithm: inferring finite automata models

Active concept learning of regular languages, represented as finite automata, is often just called active automata learning. In this setting, the domain is $\Sigma^*$, the set of words over a finite alphabet $\Sigma$. The concept is a regular language over $\Sigma$. The most commonly used algorithm for active automata learning is $L^*$ [14], developed by Dana Angluin in the 1980s. Since then, active automata learning has been used in a number of different applications (for an overview, see [39]), and a number of different variations have been developed, for example the optimization by Rivest and Schapire [74]. In this exposition, we will describe this optimization.

The $L^*$ algorithm infers a minimal deterministic finite automaton that recognizes a particular regular language over a finite alphabet. We refer to this language as the target language. In the $L^*$ algorithm, a membership query consists in asking whether a
word $u$ is in the target language or not. The answer is either $+$, if it is, and $-$, otherwise. An equivalence query consists in asking whether a hypothesis automaton constructed by the Learner is correct or not. The answer is either yes, or a counterexample, which is a word on which the hypothesis automaton and the Teacher disagree, i.e., a word that is in the target language but rejected by the hypothesis automaton, or vice versa.

Conceptually, $L^*$ is based on the Nerode equivalence (Section 2.2), and on partition refinement. The Nerode equivalence partitions the set of words in $\Sigma^*$ into equivalence classes. The algorithm maintains an approximation of the Nerode equivalence, which is gradually refined as the algorithm progresses. Initially, the Learner assumes that all words over the alphabet are Nerode-equivalent, i.e., that there are no distinguishing suffixes. By interacting with the Teacher, the Learner progressively discovers which words can be separated by means of distinguishing suffixes. The Learner also discovers new equivalence classes. She refines her approximation of the Nerode equivalence accordingly, until she is able to build a hypothesis automaton using the Nerode-equivalence classes to represent distinct states.

**Observation table**

In order to construct a hypothesis automaton, the Learner uses an *observation table*. An observation table organizes the information obtained by the Learner from making queries, and represents the current approximation of the Nerode equivalence. Provided that the table satisfies certain properties, it can be used to directly construct an automaton. The table is maintained by the Learner and continually extended as the algorithm proceeds. Let us see how this works, using the example language $L_{abb}$ recognized by the deterministic finite automaton in Figure 2.2.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>$\epsilon$</th>
<th>b</th>
<th>ab</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$-$</td>
<td>$-$</td>
<td>$+$</td>
</tr>
<tr>
<td>a</td>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>ab</td>
<td>$+$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>b</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>aa</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>aba</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>abb</td>
<td>$+$</td>
<td>$+$</td>
<td>$-$</td>
</tr>
<tr>
<td>ba</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>bb</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

*Figure 4.1. Example of an observation table.*

Figure 4.1 shows an observation table. Rows are labeled with prefixes; columns are labeled with suffixes. The prefixes above the double bar are called short prefixes; the prefixes below the double bar, extended prefixes. The symbol $\epsilon$ denotes the empty word. Each cell contains either $+$ or $-$, denoting whether the concatenation of the corresponding prefix and suffix is in the target language or not. If a membership query has not yet been made for a particular concatenation, that cell is empty. For example,
the second row, labeled with the prefix a, denotes that the word ab is in the target language, whereas a and aab are not.

Suffixes in the table distinguish different prefixes. For example, the suffix ε distinguishes the short prefix ab from the three other short prefixes, since the cell for the prefix ab and suffix ε is labeled +, but all other cells in the column are labeled −. The suffix ab distinguishes the short prefix ε from the three other short prefixes, since the cell for the prefix ε and suffix ab is labeled +, but all other cells in the column are labeled −. The observation table thus induces an equivalence relation on prefixes with respect to the set of suffixes; this is an overapproximation of the Nerode equivalence.

The table can be used to build a minimal deterministic finite automaton, in which the short prefixes will become states, and the extended prefixes used to define transitions. For example, the short prefix ab will become a state \( q \), and the extended prefix aba defines the state to which the automaton will move from \( q \) when processing the symbol a. The table in Figure 4.1 can be used to construct the automaton in Figure 2.2.

We are now ready to formally define an observation table.

**Definition 10** (Observation table). An observation table is a tuple \( O = (S,E,T) \) where

- \( S \) is a prefix-closed set of short prefixes,
- \( E \) is a set of suffixes, and
- \( T : (S \cup (S \cdot \Sigma)) \times E \mapsto \{+,-\} \) is a labeling function.

In an observation table, \( T(s,e) \) denotes the label for the word \( se \), i.e., whether \( se \) is a word in the target language or not.

Recall that the observation table \( O \) induces an equivalence relation \( \equiv_O \) on prefixes. Two prefixes \( s \) and \( s' \) are equivalent, denoted \( s \equiv_O s' \), if \( T(s,e) = T(s',e) \) for all suffixes \( e \) in the table. During the run of the algorithm, two short prefixes will never be equivalent. An observation table is **closed** if all prefixes have an equivalent short prefix. This property can be formalized in the following definition:

**Definition 11** (Closed observation table). An observation table \( O = (S,E,T) \) is closed if for any prefix \( s' \in (S \cup (S \cdot \Sigma)) \), there is a short prefix \( s \in S \) such that \( s \equiv_O s' \).

An observation table that is not closed has a prefix \( s' \) for which there is no equivalent short prefix. The table can be made closed by promoting \( s' \) to a short prefix.

**Definition 12** (Constructing a finite automaton from an observation table). A closed observation table \( O = (S,E,T) \) can be used to construct a finite automaton \( A = (\Sigma,Q,q_\epsilon,F,\delta) \), in the following manner:

- \( \Sigma \) is the input alphabet.
- The set \( Q \) of states is obtained from the set \( S \) of short prefixes: each short prefix \( s \in S \) represents a state \( q_s \in Q \).
- The initial state is \( q_\epsilon \), represented by the short prefix \( \epsilon \).
- The set \( F \) consists of the accepting states in \( Q \). A state \( q_s \in Q \) is accepting whenever \( T(s,\epsilon) = + \), and rejecting otherwise.
- The set \( T \) of transitions is extracted from the set of prefixes, so that for all \( s \in S \) and \( a \in \Sigma \), we get \( (q_s,a,q_{s'}) \in T \) where \( s' \) is the unique short prefix such that \( sa \equiv_O s' \).
The phases of $L^*$

Descriptions of the $L^*$ algorithm usually separate it into three phases: hypothesis construction, hypothesis validation, and counterexample processing. These phases are iterated in order until a correct final model is obtained.

**Hypothesis construction.** The Learner constructs words corresponding to entries in the observation table that have not been filled yet, and makes membership queries for them. The Teacher’s answers are entered into the observation table. The Learner continues making membership queries, occasionally promoting extended prefixes to short prefixes until the table is closed, at which time she constructs a hypothesis automaton.

**Hypothesis validation.** The Learner submits the hypothesis automaton to the Teacher for an equivalence query. In theory, the Teacher can easily answer this question by YES or NO, since it already knows whether the model is correct or not. In practice, teachers with complete knowledge of the target language are often not available, so other methods are used.

**Counterexample processing.** If the model is not correct, the Teacher produces a counterexample. A counterexample is a word $c = a_1 \ldots a_m$ on which the Learner and Teacher do not agree, i.e., a word that is accepted by the hypothesis automaton, but which is not in the language (or vice versa). Processing the counterexample will find a suffix that, when added to the observation table, will distinguish a new state in the hypothesis automaton. Let us see how this is done.

For each $i$ such that $0 \leq i \leq m$, the prefix $a_0 \ldots a_i$ of the counterexample leads to a state in the hypothesis. We use $u_i$ to denote the short prefix that represents this state. The short prefix $u_0$ is $\epsilon$. From the short prefix $u_i$, we obtain $u_{i+1}$ as the short prefix that is equivalent to the prefix $u_ia_{i+1}$, i.e., for which $u_{i+1} \equiv_H u_i a_{i+1}$. Since the table is closed, such a prefix $u_{i+1}$ must exist. We define $v_i$ as the suffix $a_{i+1} \ldots a_m$ of the counterexample.

We observe that there must be an index $j$ such that exactly one of the words $u_{j-1}a_jv_j$ and $u_jv_j$ is in the language. If this were not the case, i.e., if for all $j$, either $u_{j-1}v_{j-1}$ and $u_jv_j$ would both be in the language or both not in the language, then either none or both of $u_0v_0$ and $u_mv_m$ would be in the language, meaning that $c$ would not be a counterexample. At the index $j$ where the counterexample and hypothesis diverge, the suffix $v_j$ is a distinguishing suffix for the prefixes $u_{j-1}a_j$ and $u_j$.

To resolve the counterexample, we add $v_j$ to the table as a new suffix. This will make $u_{j-1}a_j$ inequivalent to $u_j$, promoting $u_{j-1}a_j$ to a short prefix.

**Example: Inferring a small deterministic finite automaton**

We illustrate $L^*$ by applying it to the regular language $L_{abb}^*$, recognized by the deterministic finite automaton in Figure 2.2. The Learner initializes the observation table by adding a prefix row, labeled $\epsilon$. To complete the table, she also makes membership queries for $\epsilon$ concatenated with each alphabet symbol, adding rows for $a$, and $b$. The Learner also adds one suffix column, labeled $\epsilon$. Then, she makes membership queries to fill the table. All membership queries are answered with $\neg$. The resulting table is closed, and from it the Learner constructs the first hypothesis automaton $\mathcal{H}_1$. The hypothesis and corresponding table are shown in Figure 4.2.
The model is not correct, so the Teacher provides a counterexample, ab, which is in the target language but rejected by $\mathcal{H}_1$. Figure 4.3 shows step by step how the Learner processes the counterexample:

- The hypothesis starts running the counterexample in the state represented by the short prefix $u_0$, i.e., $\epsilon$. At index 1 of the counterexample, we consider the prefix $a$, which also leads to the state in the hypothesis represented by the short prefix $u_1 = \epsilon$. At this point, we have the suffix $v_1 = b$. Here, we see a discrepancy between the counterexample and the hypothesis: the word $u_0a_1v_1 = ab$ is in the target language, but the word $u_1v_1 = b$ is not. We have found an undiscovered location.
- The Learner adds $b$ to the set of suffixes, and makes membership queries to fill the table. This causes it to be no longer closed, since the extended prefix $a$ is not equivalent to the only short prefix, $\epsilon$ (see Figure 4.3(a)).
- The table is closed by making $a$ a short prefix, and making membership queries for $aa$ and $ab$. This again causes the table to be not closed, since the extended prefix $ab$ is not equivalent to any of the short prefixes (see Figure 4.3(b)).
- The table is closed by making $ab$ a short prefix, and making membership queries for $aba$ and $abb$.

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
Prefix & Suffixes \\
\hline
$\epsilon$ & $b$ \\
$\epsilon$ & $-$ \\
a & $-$ \\
b & $-$ \\
\hline
\end{tabular}
\hspace{1cm}
\begin{tabular}{|c|c|}
\hline
Prefix & Suffixes \\
\hline
$\epsilon$ & $-$ \\
an & $+$ \\
b & $-$ \\
aa & $-$ \\
ab & $+$ \\
\hline
\end{tabular}
\caption{Not closed observation tables.}
\end{figure}

(a) No short prefix is equivalent to $a$. (b) No short prefix is equivalent to $ab$. 

*Figure 4.2.* Learner’s hypothesis $\mathcal{H}_1$ and corresponding observation table.
The Learner now constructs her second hypothesis automaton $H_2$. The hypothesis and corresponding table are shown in Figure 4.4. This hypothesis is also not correct, since it does not distinguish the states represented by $\epsilon$ and $b$. A counterexample will prompt the addition of the suffix $ab$, resulting in the table shown in Figure 4.1. This observation table correctly represents the target language, so the Teacher will answer YES to the last equivalence query. The final automaton is shown in Figure 2.2.

**Complexity of $L^*$**

The $L^*$ algorithm is guaranteed to converge and produce the minimal deterministic finite automaton that recognizes the target language. If $n$ is the number of states in the minimal automaton, $m$ the length of the longest counterexample provided by the Teacher, and $k$ the size of the input alphabet, the upper bound on the number of equivalence queries is $n$, and on the number of membership queries, $kn^2 + n \log m$, by the following reasoning.

- Each counterexample will trigger the addition of one new suffix (column) to the observation table, which will distinguish two prefixes and generate a new state. Thus, the upper bound on the number of equivalence queries is $n$.
- Membership queries are used for two purposes: to close the table and to process the counterexample. A table is closed by making $k$ membership queries for one-symbol extensions of a short prefix, which can happen at most $n$ times, so the upper bound is $kn^2$. To process the counterexample according to the approach by Rivest and Schapire [74], we find a distinguishing suffix using binary search. This requires $\log m$ membership queries. Since there may be at most $n$ counterexamples, the upper bound is $n \log m$. In total, the upper bound on the number of membership queries is $kn^2 + n \log m$. 

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$\epsilon$, $b$</td>
</tr>
<tr>
<td>$a$</td>
<td>$-$, $+$</td>
</tr>
<tr>
<td>$ab$</td>
<td>$+$, $+$</td>
</tr>
<tr>
<td>$b$</td>
<td>$-$, $-$</td>
</tr>
<tr>
<td>$aa$</td>
<td>$-$, $-$</td>
</tr>
<tr>
<td>$aba$</td>
<td>$-$, $-$</td>
</tr>
<tr>
<td>$abb$</td>
<td>$+$, $+$</td>
</tr>
</tbody>
</table>
4.2 The $SL^*$ algorithm: inferring register automata models

The $SL^*$ algorithm (Paper III) is an adaptation of the $L^*$ algorithm to the symbolic setting, by which we mean that the algorithm can be used to infer register automata models of data languages. Like $L^*$, $SL^*$ follows the MAT model and is based on interactions between a Learner and a Teacher. An important aspect in which $SL^*$ differs from $L^*$ and $SL^*$ is the types of queries that the Teacher can answer. Instead of membership queries, the Teacher in the $SL^*$ algorithm answers tree queries by means of a tree oracle (Section 3.2). When the Teacher answers a tree query, she actually forwards the query to the oracle, who makes a number of membership queries and generates a symbolic decision tree from these queries and their answers. The symbolic decision tree is then returned to the Learner.

Conceptually, $SL^*$ is based on the symbolic Nerode equivalence (Section 3.2) and on partition refinement. The Learner’s initial assumption is that all data words are equivalent. By interacting with the Teacher, the Learner progressively discovers which prefixes can be separated by means of symbolic suffixes, in the sense that the tree oracle returns inequivalent symbolic decision trees for them. Initially, the Learner makes tree queries that return symbolic decision trees for shorter suffixes; she then refines the trees by adding more suffixes to the tree queries. The approximation of the symbolic Nerode equivalence is refined accordingly, until the Learner can build a hypothesis automaton where Nerode-equivalence classes represent locations. Transitions, guards, and assignments in the automaton are generated from the symbolic decision trees.

Observation table

As in the $L^*$ algorithm, the Learner maintains an observation table, which organizes the information the Learner has obtained from making queries. The table represents the current approximation of the symbolic Nerode equivalence. It can be used to directly construct a register automaton, provided that it satisfies certain properties. The table is continually improved as the algorithm proceeds, though in a slightly different way than in $L^*$: symbolic decision trees are used in place of suffixes, but instead of adding new trees, existing trees are refined by increasing the set of symbolic suffixes used to construct them. We illustrate how this works for the language $L_{ra}$.

Figure 4.5 shows an observation table used by the $SL^*$ algorithm. As in $L^*$, the table has rows. However, it has only one column, labeled with a set $V$ of symbolic suffixes. Each cell contains the $(u,V)$-tree constructed by the canonical tree oracle for the prefix $u$ with which the row is labeled, and the set $V$ of symbolic suffixes with which the column is labeled. For example, the cell in the second row, labeled $a(2)$, contains the tree $O(a(2),\{\epsilon,a,aa\})^1$.

Just as in $L^*$, the prefixes in the table are divided into short and extended prefixes. Each extended prefix is equivalent to one short prefix. When a symbolic decision tree in the lower part of the table is refined, it may become inequivalent to all symbolic decision trees in the upper part of the table. Whenever this happens, the associated prefix is promoted to a short prefix.

---

1This is actually the same tree as in Figure 3.5, rotated 90 degrees.
<table>
<thead>
<tr>
<th>Prefix</th>
<th>Symbolic suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$a(p)$, $a(p)$</td>
</tr>
<tr>
<td>$a(2)$</td>
<td>$a(p) \mid x_1 \leq p$, $a(p)$</td>
</tr>
<tr>
<td></td>
<td>${ x_1 }$ $a(p)$</td>
</tr>
<tr>
<td></td>
<td>$a(p) \mid p &lt; x_1$, $x_2 := p$</td>
</tr>
<tr>
<td></td>
<td>$a(p) \mid x_2 \leq p$</td>
</tr>
<tr>
<td></td>
<td>$a(p)$</td>
</tr>
<tr>
<td>$a(2)a(1)$</td>
<td>$a(p) \mid x_1 \leq p$, $a(p)$</td>
</tr>
<tr>
<td></td>
<td>${ x_1 }$ $a(p)$</td>
</tr>
<tr>
<td></td>
<td>$a(p) \mid p &lt; x_1$, $x_2 := p$</td>
</tr>
<tr>
<td></td>
<td>$a(p) \mid x_2 \leq p$</td>
</tr>
<tr>
<td></td>
<td>$a(p)$</td>
</tr>
<tr>
<td>$a(2)a(1)a(0)$</td>
<td>$a(p)$</td>
</tr>
<tr>
<td>$a(2)a(3)$</td>
<td>(same as $a(2)$, with $x_1$ renamed to $x_2$)</td>
</tr>
<tr>
<td>$a(2)a(1)a(3)$</td>
<td>(same as $a(2)$, with $x_1$ renamed to $x_3$)</td>
</tr>
<tr>
<td>$a(2)a(1)a(0)a(3)$</td>
<td>(same as $a(2)a(1)a(0)$)</td>
</tr>
</tbody>
</table>

$V = \{ \epsilon, a, aa \}$

Figure 4.5. Example of an observation table.

For each initial transition of a $(u,V)$-tree in the upper part of the table, there is an extended prefix in the lower part of the table. For example, the $(u,V)$-tree for $u = a(2)a(1)$ has a register $x_2$ that stores the second data value from the prefix $a(2)a(1)$, i.e., 1. The tree has two initial transitions, guarded by $x_2 \leq p$ and $p < x_2$, respectively. Each of these transitions will give rise to a new prefix, i.e., we will get two extended prefixes of the form $u\alpha(d_g^u)$, with $d_g^u$ being the representative data value (see Definition 7) for each guard $g$ after $u$:

- From the transition guarded by $x_2 \leq p$, we get $d_g^u = d_{a(2)a(1)}^{x_2 \leq p}$.
  
  We can define $d_g^u$ as 3, so the resulting extended prefix is a $(2) a(1) a(3)$.

- From the transition guarded by $p < x_2$, we get $d_g^u = d_{a(2)a(1)}^{p < x_2}$.
  
  We can define $d_g^u$ as 0, so the resulting extended prefix is a $(2) a(1) a(0)$.

Note that in the observation table in Figure 4.5, $a(2)a(1)a(0)$ has actually been promoted to a short prefix, whereas $a(2)a(1)a(3)$ remains in the lower part of the table. This is because the $(u,V)$-tree for $u = a(2)a(1)a(0)$ is inequivalent to all other $(u,V)$-trees in the upper part of the table.

We define an observation table for $SL^*$ as follows:

**Definition 13** (Observation table). An observation table $O$ for $SL^*$ is a tuple $O = (U,U^+,V,Z)$ where
• $U$ is a prefix-closed set of short prefixes,
• $U^+$ is a set of extended prefixes, each of the form $ua(d)$
• $V$ is a set of symbolic suffixes, and
• $Z$ maps each prefix $u \in (U \cup U^+)$ to a $(u, V)$-tree.

The set $U^+$ of extended prefixes contains exactly those words $ua(d)$ where $d$ is the representative data value $d_u^0$ for a guard $g$ on an initial branch in $Z(u)$.

In an observation table, the Learner fills the cells $Z(u)$ with $(u, V)$-trees generated by an oracle. The observation table induces an equivalence relation on prefixes: two prefixes $u$ and $u'$ are equivalent if $Z(u) \equiv_\gamma Z(u')$ for some renaming $\gamma$ of registers in the root locations of the trees. An observation table is closed if for any extended prefix $u \in U^+$, there is a short prefix $u' \in U$ such that $Z(u) \equiv_\gamma Z(u')$ for some renaming $\gamma$ of registers. This is exactly the same concept as in $L^*$, except individual suffixes are replaced here by symbolic decision trees.

An observation table for register automata must also be able to take into account assignments to registers. To do this, we need the following definition:

**Definition 14 (Register-consistent observation table).** An observation table is register-consistent if for each extended prefix $ua(d) \in U^+$, whenever $d_i$ is a data value in $u$ and $x_i \in X(Z(ua(d)))$, then there is also a register $x_i \in X(Z(u))$.

Register-consistency is a new property specific to the register automata setting. The $SL^*$ algorithm needs this property to guarantee that data values that are needed later in a run of the register automaton are stored when they first occur. Intuitively, if an observation table is not register-consistent, there is a relation between a data value in a suffix and a data value $d_i$ in a prefix, which appears in a guard in a symbolic decision tree, but there is no register $x_i$ that can store $d_i$.

A table is made register-consistent by adding the symbolic suffix $\alpha v$ to $V$, where $v$ refers to the symbolic suffix that causes $x_i$ to appear in a guard. After adding $\alpha v$, $Z(u)$ will contain an initial transition for $\alpha (d)$, and the value $d_i$ that occurs in $u$ will be stored in the register $x_i$.

**Definition 15 (Constructing a register automaton from an observation table).** A closed and register-consistent observation table $O = (U, U^+, V, Z)$ can be used to construct a register automaton $\langle L, l_0, F, X, \Gamma \rangle$, according to the following specification.

• The set $L$ of locations is obtained from the set $U$ of short prefixes: each short prefix $u \in U$ represents a location $l_u \in L$.
• The initial location $l_0$ is represented by the short prefix $\epsilon$.
• The set $F$ consists of the accepting locations in $L$. A state $l_u \in L$ is accepting whenever the root node of the tree $Z(u)$ is accepting, and rejecting otherwise.
• For each location $l_u \in L$, the registers $X(l)$ are obtained as the registers in the root node of $Z(u)$.
• The transitions are extracted from the set of short prefixes and their symbolic decision trees. Each initial transition $\langle l, \alpha(p), g, \pi, l' \rangle$ in $Z(u)$ generates a transition $\langle u, \alpha(p), g, \pi', l' \rangle$ where
  - $u'$ is the unique short prefix in $U$ such that $ua(d^0_u) \equiv_\gamma u'$,
  - $\alpha(p)$ is a parameterized input symbol with the parameter $p$. 

46
- $g$ is a guard over $p$ and $X(l)$ (the same guard as in $Z(u)$, and
- $\pi'$ the assignment, generated from $\pi$ and $\gamma$, i.e., it is a mapping from $X(Z(u'))$ to $X(Z(u))$ and $p$. It assigns some data value to the register $x_i \in X(Z(u'))$: either $\gamma^{-1}(x_i)$ if the already stored value is kept, or $p$ if the current input parameter is stored). □

When building a register automaton from the table, the short prefixes will be used as locations, since they are mutually inequivalent. The extended prefixes will be used to represent transitions: if $u \in U$ is a short prefix (which will become a location) then, when processing a data symbol $\alpha(d)$ that satisfies a guard $g$ in one of the initial transitions of $Z(u)$, the automaton will move to the location represented by $u \alpha(d)g$.

The guards and assignments on each initial transition of a $(u, V)$-tree will thus become guards and assignments in the register automaton.

For example, the short prefix $a(2)$ will become a location $l$ in a register automaton constructed from the observation table in Figure 4.5. The tree $Z(a(2))$ has two initial transitions, guarded by $x_1 \leq p$ and $p < x_1$, respectively. Each of these is represented as a prefix in the table: $a(2)a(1)$, which is also a short prefix, and $a(2)a(3)$, which is an extended prefix. Thus, we create two transitions from $l$, both of form $\langle l, \alpha(p), g, \pi, l' \rangle$:

- The guard $g$ is $p < x_1$. The corresponding prefix is $a(2)a(1)$, which is a short prefix. In the tree, we have $x_2 := p$, so the assignment $\pi$ is $\pi(x_2) = p$. The location $l'$ is the location represented by $a(2)a(1)$.
- The guard $g$ is $x_1 \leq p$. The corresponding prefix is $a(2)a(3)$, for which $Z(a(2)a(3)) \equiv_Z Z(a(2))$ whenever $\gamma(x_2) = x_1$. In the tree, we have no assignment (which means we reassign registers). Thus, we obtain the assignment $\pi(x_1) = \gamma^{-1}(x_1) = x_2$. The location $l'$ is the location represented by $a(2)$.

The phases of $SL^*$

Similar to $L^*$, we also separate the description of $SL^*$ into three phases: hypothesis construction, hypothesis validation, and counterexample processing. These phases are iterated in order until a correct final model is obtained.

**Hypothesis construction.** The Learner makes tree queries for combinations of concrete prefixes and sets of symbolic suffixes. The Teacher takes advantage of the canonical tree oracle to construct $(u, V)$-trees, which are then returned to the Learner and used to fill the observation table. The Learner continues making tree queries until the table is closed and register-consistent, which may involve adding more prefixes and more symbolic suffixes. Then, she constructs a hypothesis automaton from the table.

**Hypothesis validation.** The Learner submits the hypothesis automaton to the Teacher for an equivalence query. In theory, the Teacher can easily answer this question by YES or NO, since it already knows whether the model is correct or not. In practice, teachers with complete knowledge of the target language are often not available, so other methods are used. We offer a more detailed discussion of how to implement or mimic equivalence queries in practice, in Paper IV.

**Counterexample processing.** If the model is not correct, the Teacher produces a counterexample. A counterexample is a string on which the Learner and Teacher do
not agree, i.e., a string that is accepted by the hypothesis automaton, but the Teacher knows should not be accepted, or vice versa. The counterexample is processed in a manner similar to that described by Rivest and Schapire [74]. The result of processing a counterexample will be the addition of a new symbolic suffix to $V$. We describe counterexample processing in more detail in the next section.

**Counterexample processing in $SL^*$**

The goal of counterexample processing in $SL^*$ is to find a symbolic suffix which we will add to the set $V$ of symbolic suffixes in the observation table. Let us see how this can be done.

We assume that the Learner has constructed a hypothesis automaton $H$, submitted it for an equivalence query, and received a counterexample $c$ in return. The counterexample is a data word on which $H$ and the Teacher disagree, i.e., a word that is in the target language but rejected by $H$, or vice versa. Technically, the counterexample is a data word $c = \alpha_1(d_1) \ldots \alpha_m(d_m)$. Each prefix $\alpha_1(d_1) \ldots \alpha_i(d_i)$ leads to a state in the hypothesis that is represented by the short prefix $u_i$. We define $v_i$ as the symbolic suffix $\alpha_{i+1} \ldots \alpha_m$.

In the counterexample, there is some index $j$ where the symbolic decision tree $O_L(u_j,\{\alpha_{j+1} \ldots \alpha_m\})$ disagrees with the symbolic decision tree $O_L(u_{j-1},\{\alpha_j \ldots \alpha_m\})$ on how to classify the counterexample (i.e., one tree accepts and the other does not). If there is no such $j$, then there is no discrepancy between the counterexample and the hypothesis, in analogy with the principles for counterexample processing in $L^*$.

There are two possible reasons why the counterexample and the hypothesis would disagree: either one of the guards in the hypothesis is not refined enough, or the renaming $\gamma$ of registers used in the hypothesis is wrong. We now describe each case.

**Case (i): One of the guards in the hypothesis is not refined enough.** Let $g_j$ be the guard of the initial transition of $Z(u_{j-1})$ that $\alpha_j(d_j)$ satisfies, and $g'_j$ the guard of the initial transition of $O_L(u_{j-1},\{v_j\})$ that $\alpha_j(d_j)$ satisfies. In this case, $g'_j$ is more refined than $g_j$, which means that the symbolic decision tree $O_L(u_{j-1},\{v_j\})$ has more initial branches than the symbolic decision tree $Z(u_{j-1})$ in the observation table.

We add $v_{j-1}$ to $V$, which will result in at least one new or refined transition from $u_{j-1}$ in the hypothesis (Definition 8).

**Case (ii): The renaming $\gamma$ of registers used in the hypothesis is wrong.** Let $T$ be the subtree of $O_L(u_{j-1},\{v_j\})$ that starts in the location reached by $\alpha_j(d_{u_{j-1}}^g)$ via a transition guarded by $g_j$. We have that $T \neq O_L(u_j,\{v_j\})$, i.e., the trees are not isomorphic under the renaming $\gamma$.

We add $v_j$ to $V$, which will result in one of two possible outcomes: either the remapping used in the hypothesis will be refined, or the prefix $u_{j-1}\alpha_j(d_{u_{j-1}}^{g_j})$ will be promoted to a short prefix.

By the above arguments, we see that processing the counterexample will result in a new suffix being added to the set of symbolic suffixes in the observation table. This will refine the symbolic decision trees in the observation table, which in turn will lead to a new location or a new transition being discovered, or to the renaming of registers in some transition of the hypothesis being refined.
Example: *Inferring a small register automaton*

We describe an application of $SL^*$ to learning the register automaton in Figure 3.2. The Learner initializes the observation table by adding a prefix row, labeled $\epsilon$. She makes a tree query for $\epsilon$ and the set $\{\epsilon\}$ of symbolic suffixes. The resulting tree is added to the observation table. To complete the table, she also makes tree queries for $\epsilon$ concatenated with the only alphabet symbol, adding a row for $a(2)$. The resulting table is closed and register-consistent, and from it the Learner constructs its first hypothesis automaton $H_1$. The hypothesis and corresponding table are shown in Figure 4.6.

![Diagram of a(2) transition](image)

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Symbolic suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$V = {\epsilon}$</td>
</tr>
<tr>
<td>$a(2) \epsilon$</td>
<td>same as $\epsilon$</td>
</tr>
</tbody>
</table>

**Figure 4.6.** Learner’s hypothesis $H_1$ and corresponding observation table.

The model is not correct, so the Teacher provides a counterexample. The counterexample is $a(2) a(1) a(0)$, which is not in the target language but accepted by $H_1$. After processing the counterexample, the suffix $aa$ is added to the set of symbolic suffixes. This causes the table to be not closed, because the tree $Z(a(2))$ is no longer isomorphic to the tree $Z(\epsilon)$. The prefix $a(2)$ is promoted to a short prefix.

The Learner makes tree queries for the extended prefixes corresponding to each of the initial transitions in $Z(a(2))$, i.e., to obtain the trees $O^L(a(2) a(1), V)$ and $O^L(a(2) a(3), V)$. The result is shown in Figure 4.7. The table is not closed, since the tree $Z(a(2) a(1))$ is not isomorphic to any of the trees in the upper part of the table. The table is closed by promoting $a(2) a(1)$ to a short prefix.

The Learner now makes tree queries for the extended prefixes corresponding to each of the initial transitions in $Z(a(2) a(1))$, i.e., to obtain $O^L(a(2) a(1) a(0), V)$ and $O^L(a(2) a(1) a(3), V)$. The table is again not closed, since $Z(a(2) a(1) a(0))$ is not isomorphic to any of the trees in the upper part of the table. The table is closed by promoting $Z(a(2) a(1) a(0))$ to a short prefix. The Learner makes a tree query for the extended prefix $a(2) a(1) a(0) a(3)$ corresponding to the only initial transition in the tree $Z(a(2) a(1) a(0))$, in order to complete the table.

The suffixes in $V$ distinguish the four locations $l_0, l_1, l_2$ and $l_3$ in Figure 3.2. In the observation table in Figure 4.5, location $l_0$ is represented by $\epsilon$, location $l_1$ is represented by $a(2)$, location $l_2$ is represented by $a(2) a(1)$ and the sink location $l_3$ is represented by $a(2) a(1) a(0)$. The Learner uses the table to construct a register automaton that correctly represents the target language, so the Teacher will answer YES to the last equivalence query. The final automaton is shown in Figure 3.2.
4.3 Automata learning in realistic scenarios

In Section 4.2, we described $SL^*$, a learning algorithm for register automata. We will now describe an extension to $SL^*$ that we have developed, which is oriented towards more practical applications of automata learning techniques. Recall that $SL^*$ infers register automata models of regular data languages, i.e., automata that accept or reject data words. However, many actual programs or components are not quite as simplistic: they do not simply state whether executing a command or invoking a method is OK or NOT OK; they produce output of some kind.

Figure 4.8 shows code for a small program that implements a map. The map stores key-value pairs $\langle k, v \rangle$ and exposes two methods: put and get. When we invoke put$(v)$, the map checks that its capacity is not exhausted, creates a new unique key $k$, and returns $k$ as output. When we invoke get$(k)$, the map checks that it contains a pair $\langle k, v \rangle$ and returns $v$ as output.

How can we infer a model of a component such as the map in Figure 4.8, in a manner similar to $SL^*$? One way is to invent a new register automaton model (or use an existing one) that allows for non-binary output, and a learning algorithm that can infer such models. Howar et al [51] did exactly that: they generalized the register automata in Paper I to transducers that generate output symbols in response to input symbols. They also extended the learning algorithm in Paper II to the input/output setting, but they only consider the theory of equality.

Another way, which we will explore here, is to not invent a new formalism, but reuse the register automata formalism from Paper III and treat sequences of alternating input and output as words in a data language. In this manner, we can treat the
class KeyGenMap {
    private Map map = new HashMap();
    K put(V val) {
        assert map.size() < MAX_CAPACITY;
        K key = generateUniqueKey();
        map.put(key, val);
        return key;
    }
    V get(K key) {
        assert map.containsKey(key);
        return map.get(key);
    }
}

Figure 4.8. Code for a map that generates keys.

problem as one of language inference, and apply the SL* algorithm without fundamental modifications.

We extend the SL* algorithm, not by altering the basic algorithm, but by implementing it in a tool together with practical extensions.

Concretely, we consider each input or output symbol to be a data symbol of the form \( \alpha(d) \). Each data value \( d \) has a particular type that determines which other values it can be compared to. In the map example, types are \( K \) (for keys) and \( V \) (for values). We write \( d^T \) for a data value of type \( T \) with \( T \in \{K, V\} \). Only values of the same type can be compared to each other, meaning that each type is associated with a particular theory. This allows for more efficient generation of symbolic decision trees, since relations between data values of different types need not be considered.

An issue with register automata in general is that they cannot model fresh data values, i.e., data values that are inequivalent to all other data values, since this would require comparing all data values in a domain and possibly using an unbounded number of registers to do so. However, register automata can be extended to model a particular restricted form of freshness, where a data value is inequivalent to all other data values that we have already seen in a word of bounded length. We take this approach to modeling freshness, and introduce a unary relation \( fr \) to represent that a data value is inequivalent to all previously seen data values.

Example: A register automaton with input and output, typed parameters, and fresh data values

Figure 4.9 shows a register automaton model of the map that generates keys, i.e., the external behavior of the component in Figure 4.8. It is limited to capacity 1, for space reasons. The register automaton is a condensed version of the actual register automaton we learn: intermediate states, where no input is accepted and only output generated, are omitted and replaced by / in a transition label. For example, the loop in location \( l_0 \) is actually inferred as a transition \( \text{get}(p^k) \rightarrow \) to an intermediate state, and
then another transition $\text{error} \xrightarrow{\_} l_0$. Transition labels are of the form $\frac{\alpha(p^T)}{\pi}$ where $\alpha(p)$ is a parameterized symbol (sometimes abbreviated $\alpha$ if there is no parameter), $T$ is the type of the parameter $p$, $g$ is a guard over $p$ and registers, and $\pi$ is an assignment. Registers also have types, so a register $x^T$ can only store a data value of type $T$.

![Diagram of a register automaton](image)

**Figure 4.9.** Model of a map that generates keys.

The register automaton processes input symbols starting in location $l_0$. For example, the word $\text{put}(a)/\text{key}(4)\text{get}(4)/\text{val}(a)$ is accepted by the automaton. This corresponds to invoking $\text{put}(a)$ and receiving 4 as output, then invoking $\text{get}(4)$ and returning $a$ as output. The word $\text{put}(a)/\text{key}(4)\text{get}(2)/\text{error}$ is also accepted by the automaton. This corresponds to invoking $\text{put}(a)$, receiving 4 as output, then invoking $\text{get}(2)$ and getting an error since no value with the key 2 is stored in the map. □
5. Summaries of papers

We summarize briefly the contents of each of Papers I-IV.

I **A succinct canonical register automaton model**

Paper I defines a register automaton model that recognizes data languages where data values can be compared for equality. It is an extended journal version of the paper [31] where the automaton model was originally defined, using slightly different but equivalent notation.

The motivation behind the paper was to come up with a formalism for register automata that was canonical, while being as succinct as possible.

At the time, different forms of (canonical) register automata had been presented, but they all suffered from restrictions such as requiring a total order on data values, or requiring stored values to be unique. These restrictions made the resulting automata unnecessarily large (in terms of state-space, and in terms of the number of registers they needed). Another reason for unnecessarily large models is encoding all equalities between data values in a data word, regardless of whether they are needed in order to recognize the data language.

In the paper, we represent a data language over the theory of equality as a classification of a set of data words. We show that any data language can be represented by restricting its classification to a finite minimal subset of **essential** data words. The essential data words exhibit exactly the equalities that are needed in order to recognize the data language. For each non-essential data word, there is an essential data word with at most as many equalities, and whose classification extends to the non-essential word.

To obtain the set of essential words for a data language, we organize data words in a tree according to two partial orders: how many equalities a word has, and at what point in the word the first equality (if any) occurs. Nodes in the tree are then merged whenever they are isomorphic, in a procedure similar to minimization of binary decision diagrams (BDDs). The resulting tree is minimal in a certain class, and the branches are the essential words for the data language.\(^1\)

The main result in the paper is a Nerode equivalence for data languages over the theory of equality. Using this equivalence, we can fold a set of essential data words into a register automaton. Two essential data words will lead to the same location in the automaton if their residual languages are equivalent, i.e., if whenever we extend both data words with equivalent suffixes, the words will remain equivalent.

The automata we construct in this manner are canonical and **determinate**: a data word may correspond to more than one path in the automaton, but the paths are either

\(^1\)The essential words correspond to the branches of the canonical symbolic decision trees in Paper III.
all accepting or all rejecting. The models can be made deterministic, but this conver-
sion can be done in several ways and will thus not result in canonical models. We
prove that a data language is recognizable by a determinate register automaton if and
only if the Nerode equivalence on the set of essential words has finite index.

We compare our formalism to others where different restrictions have been made
on the data values, and show that our models can be exponentially more succinct while
remaining canonical.

II Inferring Canonical Register Automata

Paper II presents an extension of the $L^*$ algorithm, enabling us to infer the register
automata defined in Paper I. The $L^*$ algorithm, a well-known example of active au-
tomata learning, had already been successfully used to infer control flow models of,
e.g., communication protocols. However, for models that capture both control flow
and data flow (e.g., register automata), existing automata learning frameworks were
either too simplistic, required cumbersome user-supplied abstractions on the data do-
main, or used a two-step approach where the automaton and the abstraction were
generated separately. The paper was motivated by the lack of an algorithm that could
directly infer the interplay between data flow and control flow, and generate accurate
models. The natural choice was to extend the $L^*$ algorithm to the register automata
setting.

The $L^*$ algorithm for finite automata uses the well-known Nerode equivalence as a
basis. In this paper, we used the Nerode equivalence for register automata (defined in
Paper I) as a basis for the new extended $L^*$ algorithm. The extended algorithm iter-
ates two phases: hypothesis construction, hypothesis validation, and counterexample
processing.

In the hypothesis construction phase, membership queries are made and the results
collected in an observation table. In contrast to $L^*$, membership queries are made
for sets of data words rather than individual words. Technically, we concatenate a
data word (prefix) with an abstract suffix, i.e., a data word where data parameters are
not instantiated. Then, the data parameters are instantiated with data values equal or
inequal to data values in the prefix, according to a set of constraints. The answer to
a membership query is a closure, i.e., a mapping from each constraint in the set to
ACCEPT or REJECT.

The algorithm continues to make membership queries until the observation table is
closed and register-consistent. The register-consistency criterion is new and specific
to the register automata setting. If the observation table is not register-consistent, this
means that there is an equality between a data value in a suffix and a data value in a
prefix, but there is no register that stores the data value in the prefix. In other words,
we have failed to take into account a register assignment.

When the observation table is closed and register-consistent, a hypothesis automa-
ton is generated and submitted for an equivalence query. If the hypothesis automaton
is correct, the algorithm terminates; otherwise, a counterexample is returned and pro-
cessed. The counterexample is guaranteed to induce a new location, transition, or
register.

Abstract suffixes correspond to the symbolic suffixes in Paper III.
We have implemented the algorithm in LearnLib and applied it to a fragment of the XMPP protocol for instant messaging. The inferred model needs a fraction of the states necessary for a model without registers.

III Active Learning for Extended Finite State Machines

This paper introduces a general black-box framework, $SL^*$, for inferring register automata where data values can be compared using different binary relations. It is an extended journal version of the paper [32] where the framework was first introduced. The paper also contains an updated and improved definition of the register automata for binary relations first presented in [33].

Prior to our paper, there had been several other works presenting extensions of the $L^*$ algorithm to extended finite state machines. These, however, tended to focus on only equality between data values and sometimes require additional information, such as access to source code. The motivation behind this paper was to bypass these restrictions and present a general extension of $L^*$ that could be used out of the box, and that would require as little manual assistance as possible. We wanted the algorithm to take into account both different data domains as well as different relations between data values.

In this paper, we describe how to learn register automata that recognize data languages where data values can be compared using different binary relations (not just equality). To that end, we also present a formalism for register automata that is parameterized on a theory, i.e., the combination of a data domain and one or more binary relations. The learning algorithm is a generalization of our prior algorithm (Paper II), while the automaton formalism improves the one presented in [33], and is a generalization of the one presented in Paper I.

A central concept in the $SL^*$ framework is that of tree queries. Tree queries are used to characterize and separate prefixes, much in the same way as membership queries in classic $L^*$. Making a tree query for a prefix returns a symbolic decision tree that describes a fragment of the data language after the prefix, i.e., ‘what happens’ when concatenating the prefix with different suffixes. Branches in a decision tree represent different suffixes and the nodes state whether a particular data word (prefix concatenated with suffix) is accepted or rejected. The $SL^*$ algorithm iterates the same phases as $L^*$: hypothesis construction, hypothesis validation, and counterexample processing. The observation table stores prefixes and corresponding symbolic decision trees obtained by making tree queries.

We give necessary conditions for a tree oracle, i.e., the function that generates a symbolic decision tree as an answer to a tree query. For the theory of equalities over an infinite domain (such as natural numbers) and for the theory of inequalities over rational or real numbers, we describe tree oracles and show that both fulfill the stipulated conditions. We give a Myhill-Nerode theorem for these theories. Increments and constants are also discussed and we give an idea of how to construct tree oracles when using these extensions together with a theory.

We evaluate our framework on a series of small examples. The examples (e.g., a sequence number counter, an alternating bit protocol, and a prepaid card) illustrate the different theories we can use. We also infer two larger models: the connection
establishment part of TCP and a priority queue. These indicate the utility of our framework in more realistic settings.

IV RALib: A LearnLib extension for inferring EFSMs

Paper IV introduces RALib, an extension to the LearnLib framework for automata learning. The extension consists of a stable reimplementation of the $SL^*$ algorithm (Paper III) together with a set of additional features.

While the $SL^*$ algorithm provides a general framework for inferring register automata, it was not yet fully usable in real-world scenarios. By reimplementing the algorithm, adding more practical features and packaging it as an extension to LearnLib, we wanted to make $SL^*$ more easily available and useful to the public.

In the paper, we describe several features that were added onto the $SL^*$ algorithm. With RALib, users can learn models with input and (non-binary) output. This is possible through a filter that essentially treats sequences of alternating in- and output as if they were sequences of input. An input/output sequence can then be 'accepted' or 'rejected' based on whether it is a valid sequence or not when executed on the target component. RALib also handles 'fresh' (i.e., previously unseen) data values, something which in principle is not possible in a register automaton. RALib, however, can work around this problem by considering data values 'fresh' if they do not occur in a particular generated test case.

Another important feature is typed data parameters: in RALib, each data parameter is assigned to a particular theory. This allows users to mimic the actual data types in a target component. It also allows for mixing different theories in the same model, since, e.g., parameters of one type can be compared for equality and parameters of another type for inequality ($\lt$). Finally, RALib comes with a Java class analyzer, and heuristics for finding counterexamples.

We evaluate RALib by comparing it to Tomte and LearnLib$^R$A, the two most similar available tools for inferring register automata, on two aspects: performance and practicality. We use the benchmarks presented in [5]. In the paper, we detail different optimizations and how they affect the number of tests executed during learning. In terms of the number of tests used during learning, RALib always outperformed LearnLib$^R$A and often Tomte on these benchmarks. In terms of practicality, we tested the Java class analyzer by inferring some data structures (FIFO/LIFO sets and a priority queue). Here, RALib needed a similar number of tests as the other tools. However, none of the other tools were able to infer the priority queue, since they only support data languages over the theory of equality.
6. Related work

This chapter reviews related work, along the main topics of this thesis: automata models and automata learning. We also discuss some practical uses for automata learning approaches.

6.1 Representing languages over infinite alphabets

Many different formalisms for representing languages over infinite alphabets have been proposed; an overview can be found in [77]. In these formalisms, alphabet symbols can be taken directly from an infinite domain (e.g., integers or natural numbers), but they can also be composed of actions (from a finite domain) that carry data values from an infinite domain. Languages over infinite alphabets can be used for representing, e.g., communication protocols or behavioral interfaces of data structures.

A popular way to model languages over infinite alphabets is using different types of automata, e.g., register automata or finite-memory automata [17, 57] (see also Chapter 3), timed automata [11], pebble automata [41], and data automata [22, 23]. Here, we consider automata that are similar to the register automata we defined in Chapter 3.

Finite-memory automata (register automata).

Introduced by Kaminski and Francez, finite-memory automata [57] is an early formalism often also referred to as register automata. Finite-memory automata use a sequence of ‘windows’ to store values, similar to registers in a register automaton. At each transition, the input symbol is compared to all values currently stored in the windows.

To improve the formalism, Benedikt et al [17] do not store all data values in their deterministic finite-memory automata – only those that are needed in later comparisons. Such values are called memorable. Memorable data values were also identified by Berg et al in [19], in the context of symbolic Mealy machines, a formalism that is similar to register automata but allows output.

By identifying memorable data values, Benedikt et al can use potentially fewer registers than Kaminski and Francez’s finite-memory automata, while keeping the same expressive power. For example, consider the data language over a set $\mathcal{A} = \{a, b\}$ of actions and the domain of integers, where words are in the language if the data value in a b-symbol is equal to the data value in the previous a-symbol. The word $a(1)b(1)$ is in this language, but the word $a(1)b(2)$ is not. This means that the value 1 is memorable and should be stored by the automaton after reading a, since there is a comparison between the data value in the b-symbol and the data value in the a-symbol. Deterministic finite-memory automata are canonical, due to a Myhill-Nerode-like theorem provided by the authors.
In both finite-memory automata and deterministic finite-memory automata models, the registers are ordered, and the stored data values are unique. These restrictions can, however, also often cause the automata to become unnecessarily large. We discuss this in detail in Section 6 of Paper I, but here give an idea of the implications.

The order on registers means that (assuming registers are ordered $x_1 \ldots x_n$) whenever a register $x_i$ is assigned a value on some transition in the automaton, then the register $x_{i-1}$ must have been assigned a value on some preceding transition. For example, we consider the case where data symbols can carry multiple data values, i.e., of the form $\alpha(d_1 \ldots d_k)$ for some $k$. With an order on registers, we get $k!$ different possible ways to store the values. In contrast, our register automata do not have an order on registers, so we need (at most) $k$ registers in this case.

The requirement to store unique data values means that no more than one instance of the same data value can be stored in registers (even if there are several in a data word). With this restriction, a data word $\alpha_1(d_1) \ldots \alpha_m(d_m)$ where $d_i = d_j$ will not use the same path in the automaton as a data word $\alpha_1(d'_1) \ldots \alpha_m(d'_m)$ with $d_i \neq d_j$, since in the first case, only one register can be assigned and compared to data values in future transitions, whereas in the second case, two registers are available. In contrast, our register automata do not require stored data values to be unique, which means that both words can use the same path in the automaton.

Decidability for finite-memory automata and connections to different logics have been investigated, e.g., in [28, 40, 43, 72, 75]. In [58], finite-memory automata were extended to the setting where registers can be updated nondeterministically during a run of the automaton.

Data automata.

Bojańczyk et al introduce data automata [22, 23, 24] and reason about them from an algebraic perspective. In [20], Björklund and Schwentick present the expressively equivalent class-memory automata which, unlike data automata, can be made deterministic.

A data automaton considers two components of a data word $\alpha_1(d_1) \ldots \alpha_n(d_n)$: its sequence $\alpha_1 \ldots \alpha_n$ of actions and its classes, i.e., the sets of positions in the word that have equal data values. To do this, it is essentially composed of two automata: a base automaton and a class automaton. The base automaton processes the sequence of actions in a word and accepts or rejects it. It compares data values for equality, and produces class strings, i.e., sequence of actions whose data values are equal. For example, a base automaton that accepts the data word $a(20)b(43)b(20)$ can produce the output class strings $ab$ and $b$.

The class automaton recognizes the regular language that consists of the set of output class strings produced by the base automaton. In this manner, the data automaton as a whole can be used to recognize data languages where data values are compared for equality, but they can also recognize languages where all data values are inequal: this simply corresponds to the regular language over class strings of length 1. Register automata cannot recognize these languages since that would require storing a possibly infinite number of data values in registers. Data automata and class-memory automata are thus strictly more expressive than register automata.
Symbolic automata.

A formalism that corresponds to register automata over different theories is that of symbolic automata [38, 66, 49]. Symbolic automata, however, do not use registers. They process words over infinite alphabets by using constraints over data symbols, so transitions are labeled with sets of input symbols, similar to guards in our register automata. For example, a symbolic automaton’s behavior may be determined by whether an input symbol is within a certain interval or not. This corresponds to a register automaton for the theory of inequality (<).

Symbolic transducers [83] are very similar to symbolic automata, but augmented with registers, which allows for more succinct models. They have been used, e.g., for processing regular expressions. In [38] the authors describe some minimization algorithms for symbolic transducers. However, the work on symbolic transducers has rather focused on minimization and equivalence checking algorithms, not on providing a Nerode equivalence or any canonicity properties.

Automata with fresh data values.

Fresh-register automata [81] is a formalism that focuses on recognizing ‘fresh’ data values. Fresh data values, in this sense, refers to data values that have not been seen yet. A locally fresh data value is one that is not currently stored in a register; a globally fresh data value is one that has not been stored in any register so far. Fresh-register automata can capture both local freshness and global freshness. For example, a fresh-register automaton can recognize the data language of words $\alpha_1(d_1) \ldots \alpha_k(d_k)$ where for any pair $d_i, d_j$ of data values such that $i \neq j$, we have $d_i \neq d_j$. This language cannot be recognized by a register automaton.

Finite-memory automata and our register automata are a less expressive subclass of fresh-register automata, since they can capture local freshness but not global freshness. Session automata [25, 26] are another less expressive subclass, which can capture global freshness but not local freshness. Session automata are thus incomparable to register automata.

Session automata can either store a fresh data value, or read a data value from a register. Similar to the representative data words we use in Paper III, Bollig et al [26] define a symbolic normal form for data words, representing words that are indistinguishable by the equality relation and freshness. A data word on this form represents a unique set of concrete data words (similar to the symbolic automata). A set of words on symbolic normal form is a regular language if the words use only a bounded number of registers. Based on this, the authors define canonical session automata.

Another form of automata that is incomparable to our register automata is variable automata [47]. These are extensions of (nondeterministic) finite automata that can be used to model languages over infinite alphabets. They are strictly less expressive than nondeterministic finite-memory automata [58]. In a variable automaton, variables are only assigned once and cannot change their values during a run. A nondeterministic variable automaton can assign values to variables nondeterministically. Variable automata can be used to recognize, e.g., data languages over the theory of equality, or data languages where each alphabet symbol appears in each word. The authors of [47] propose variable automata as a simpler alternative to e.g., register automata or
pebble automata, since many constructions and algorithms for nondeterministic finite automata also apply to variable automata.

6.2 Learning extended finite state machines with $L^*$

In Chapter 4, we described the most common algorithm for inferring finite state machines using active learning: $L^*$. In this section, we describe different tools and frameworks that make use of the algorithm in order to infer extended finite state machine models of components, programs, or interfaces. Most approaches are white-box, meaning that domain knowledge, manual abstractions, and symbolic execution can all be used, since the source code of a target component is accessible. Other approaches (including our own) are black-box, meaning that the target component is presumed to be (almost) completely unknown and the source code is unavailable. In black-box settings, we only have access to a component’s interface. An overview of approaches that are most similar to our $SL^*$ algorithm can be found in [54].

**Black-box methods.**

Aarts et al have developed a technique [1, 6] for inferring finite-state models of real components or programs using an approach based on the $L^*$ algorithm. The technique has been implemented in a tool called Tomte. A mapper is placed as an extra layer between the learning algorithm and the target component. In principle, the mapper allows Tomte to infer models where data values are compared using different relations, as long as these are defined in the mapper. The mapper thus provides a finite-state abstraction of an infinite-state system. Most case studies using Tomte, however, have focused on the theory of equality.

Tomte has been used to infer models of protocol entities [7], the biometric passport [8], and models of bank cards [2]. In these settings, mappers were manually constructed. For certain theories, Tomte can also be used without previously constructed mappers [4].

The latest version of Tomte was recently [3] extended with support for inferring ‘fresh’ data values in a component’s output, i.e., values that have not been previously processed by the component as input.

Berg et al [19] take a different approach to inferring extended finite state machines using $L^*$. They infer models in two steps. First, they infer a Mealy machine where data values are taken from a small domain. A Mealy machine is a finite state machine with output that depends both on input and on the machine’s current state. Then, they fold this model into a symbolic Mealy machine. A symbolic Mealy machine also has input and output, but its transitions are labeled with parameterized symbols, guards, and assignments – similar to those in our register automata. Lorenzoli et al [65] also use a two-step approach, but instead combine it with passive learning. Ghezzi et al [44] use a two-step approach to infer models of Java classes by observing their behavior with a small domain of data values, and then using graph transformation to obtain an abstraction.

Howar et al [51] infer behavioral interfaces of data structures, modeling them as register Mealy machines, which are similar to symbolic Mealy machines. The authors
use an adapted version of the algorithm for learning register automata over equality that was presented in Paper II.

Maler and Mens [66] develop a symbolic version of the \( L^* \) algorithm in order to learn symbolic automata. As in the \( SL^* \) algorithm, counterexamples do not necessarily trigger the addition of a new state, but a counterexample can also be used to modify the boundaries of the subsets of the alphabet that are used as transition labels. Their approach is similar to that used by Howar et al [52], where counterexamples are used to refine alphabet abstractions when learning Mealy machine models over large alphabets. Maler and Mens’s results indicate that for very large alphabets, their algorithm is more efficient than \( L^* \); we noted a similar result when comparing the performance of register automata learning to \( L^* \) in Paper II.

Bollig et al [26] present a learning algorithm that infers the canonical session automaton for a given data language, based on the fact that the language recognized by a session automaton can be represented as a regular set of symbolic data words. They infer session automata using a version of Rivest and Schapire’s optimization of \( L^* \).

Grinchtein and Jonsson [46] extend the \( L^* \) algorithm to timed automata and introduce event-recording automata. These are a restricted form of timed automata whose deterministic form (deterministic event-recording automata) allow for the definition of a Nerode equivalence. This, in turn, makes it possible to infer them using \( L^* \). In a manner similar to our tree oracle (Section 3.2), there is an Assistant between the Learner and Teacher, who mediates by making several membership queries for each timed query that the Learner makes for a timed word.

Shahbaz et al [78] infer parameterized finite state machines that do not contain registers, but where transitions are labeled with sets of symbols. This is similar to the symbolic automata discussed in Section 6.1, and to the approach in [18], where the authors infer equivalence classes of input symbols. To complete their learning framework, Shahbaz et al develop a test generation strategy that allows them to find counterexamples using integration testing, so as to avoid equivalence checking of the model.

**White-box methods.**

In the \( Sigma^* \) framework [27], the \( L^* \) algorithm is used to infer symbolic lookback transducers (SLT). An SLT is a limited form of symbolic transducers, where the generated output may only depend on the \( k \) last seen input symbols. Since the framework is white-box, symbolic execution is used to answer membership queries and to discover the input alphabet. The Teacher maintains an abstraction (a separate SLT) representing an overapproximation of the target component or program, which is used to answer equivalence queries: the abstraction is checked for equivalence against the Learner’s hypothesis. If a counterexample is produced, this either means that the hypothesis is wrong or that the abstraction must be refined. \( Sigma^* \) has been used, e.g., for verifying web sanitizers. A limitation of \( Sigma^* \) is that it only works in scenarios where it is sufficient to consider the \( k \) most recent input values.

Psyco [45] is a white-box framework that combines an implementation of \( L^* \) with symbolic execution for generating temporal component interfaces. The authors use \( L^* \) to generate guards over method parameters and to determine when existing guards must be refined. They use three regular languages to represent legal/illegal/unknown method traces, and keep track of current method guards in a map. To answer a mem-
bership query, the Teacher exercises all possible paths through each method in the query, by satisfying the guards, answering true/false/unknown depending on whether the particular path is legal, illegal, or unknown. The three languages are modeled as a three-valued finite automaton that captures the ordering between method invocations, i.e. that represent what sequences of method invocations are legal and what sequences are not.

Xiao et al [85] infer finite-state models of Java classes using an extended version of the $L^*$ algorithm. Transitions are annotated with method calls and guards over data fields taken directly from the inferred Java classes. Guards are refined whenever the output generated by a particular method depends on some data parameter, indicating that this relation also must be modeled. The refinement is done using binary classification with support vector machines (SVM). The authors use a random walk algorithm for simulating equivalence queries through testing. Alur et al [10] also deal with Java classes; they use predicate abstraction to generate an abstract specification of a Java class and then apply active automata learning to obtain a behavioral interface.
7. Conclusion(s)

In this thesis, we have presented work in two main areas: automata theory, and algorithms for automata learning. We outline our contributions and conclusions for each area separately, followed by a section on future work.

Automata theory
In this area, our main goals were to adapt register automata to recognize a wider class of languages (i.e., by not restricting relations on data values to just equality), and to obtain canonicity by means of a Nerode equivalence for our register automata.

Paper I introduced a register automaton model that recognizes data languages where data values can be compared for equality. It is equivalent, in terms of expressivity, to other similar formalisms. The advantages of our register automata is that they are canonical, and that they can be exponentially more succinct compared to other formalisms. In the paper, we showed that our models are minimal under certain restrictions. An important contribution in this paper is a Nerode equivalence and Myhill-Nerode theorem for the class of data languages over the theory of equality.

The automaton model in Paper III is a generalization of that in Paper I in that it is parameterized on a theory, e.g., a set of binary relations together with a data domain. (The model in Paper I corresponds to our register automaton model for the theory of equality over some data domain.) An important contribution in this paper is a Nerode equivalence and a Myhill-Nerode theorem for data languages over certain theories, e.g., equality on any domain, and inequality (<) on certain domains (such as real numbers).

Algorithms for automata learning
In this area, our main goals were to adapt an active automata learning algorithm ($L^*$) to learning register automata. The first sub-goal was to be able to infer the models in Paper I, where data values can only be compared for equality. Then, we wanted to extend the approach to also infer the models in Paper III, where data values can be compared using binary relations. In addition, we wanted to provide a tool that would allow us to learn real programs using our learning algorithms.

In Paper II, we extended the $L^*$ algorithm to be able to infer the register automata models presented in Paper I, i.e., where data values can be compared for equality. We implemented our algorithm in LearnLib and used a small example to compare its performance to classic $L^*$ with an abstraction on data values. The results were positive, and showed that with our approach, much fewer states were needed. An important contribution in this paper is that it is the first active automata learning algorithm to be able to infer register automata using an $L^*$-like approach.
Paper III presents a generalization (\(SL^*\)) of the algorithm in Paper II to learning register automata that are parameterized on a theory, i.e., the models also introduced in Paper III. The theoretical basis for the new algorithm is our symbolic Nerode equivalence and Myhill-Nerode theorem. We implemented this algorithm prototypically in LearnLib and illustrated its versatility using a series of small examples that showcased how it could be used to infer register automata on different theories. An important contribution in this paper is that it is the first active automata learning algorithm to be able to infer register automata where data values are compared for more relations than just equality.

Using automata learning in practice.

In Paper IV, we introduced RALib, an extension to the LearnLib framework for active automata learning. RALib makes the \(SL^*\) algorithm more useful in realistic scenarios, by adding several new features: the ability to infer models with input and output, typed data parameters, and mixing several theories in the same model. RALib also adds functionality for inferring models with 'fresh' (i.e. previously unseen) data parameters for data languages over the theory of equality. With RALib’s built-in Java class analyzer, users can directly infer models of Java classes.

We compared RALib to other similar tools, and showed that RALib often can be more efficient in terms of performance, by using fewer queries to infer the same models. RALib also provides more features: e.g., other tools only learn data languages over the theory of equality, and none of them can handle data types.

7.1 Future work

There are some directions for future work that we would like to explore, and we outline them here.

The register automata formalisms we have defined (Papers I and III) are not always optimal for modeling, e.g., data structures. For example, when modeling the behavior of a container of size \(n\), our automata will need \(n\) registers. An interesting direction would be to develop a (register) automaton formalism that can specify the behavior of a container such as a stack using a smaller number of registers. The crucial property for the stack is that values are output in reverse input order. This behavior can be specified using only two registers, if registers are allowed to carry arbitrary data values. Such automata have been used for verification of stack implementations in [9].

The \(SL^*\) algorithm (Paper III) could be optimized in several ways. One possible optimization would be to investigate the trade-off between membership/tree queries and equivalence queries. Typically, fewer membership/tree queries result in intermediate hypotheses that are not very accurate, so the algorithm needs more equivalence queries. Since equivalence queries are more difficult to implement than membership/tree queries, it might be interesting to try to shift the balance towards more accurate intermediate hypotheses. We already exploit counterexamples quite efficiently by reusing the same one several times whenever possible, but we could try to find better counterexamples, i.e., ones that expose as many discrepancies as possible between the hypothesis and the target language.
The symbolic decision trees and tree oracles could probably be used more efficiently. Currently, the tree oracle generates symbolic decision trees that describe a fragment of a data language after a particular prefix. The trees are used primarily in three ways: they are compared to other trees for isomorphism under some renaming, they are refined (i.e., more branches are added), and they form the basis for constructing register automata. However, it is not always the case that we need all branches and nodes in a symbolic decision tree for determining whether it is equivalent to another tree, or for using it to generate a register automaton. Sometimes, two trees only differ in one or two branches or leaf nodes. In these cases, it would probably be more efficient to just compare these parts of the trees in order to see whether they are equivalent. We would like to investigate whether it is possible to improve the tree oracle so that it will only generate parts of a tree that are important or significant in some sense, or apply some sort of pruning strategy to the already generated trees.

Finally, there are some obvious ways to extend the RALib tool (Paper IV). One interesting thing would be to apply it to more case studies, particularly larger and more complex components, to study the performance of the tool. Another thing would be to automate the choice of parameter types and theories in a model. Currently, these are determined by the user based, in some sense, on prior knowledge of the target component. To improve RALib, we would want to investigate different ways of automatically inferring what theories are needed to model a particular target component. This could be achieved perhaps by some intelligent test case selection, or using counterexample-driven approaches to find an appropriate theory.
References


Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1311

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)