Active learning of interface programs

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Falk M. Howar

Dortmund
2012

Tag der mündlichen Prüfung: 30.03.2012
Dekan/Dekanin: Prof. Dr. Gabriele Kern-Isberner

Gutachter/Gutachterinnen:
Prof. Dr. Bernhard Steffen
Prof. Dr. Karl Meinke

Abstract

Computer systems today are no longer monolithic programs; instead they usually comprise multiple interacting programs. With the continuous growth of these systems and with their integration into systems of systems, interoperability becomes a fundamental issue. Integration of systems is more complex and occurs more frequently than ever before. One solution to this problem could be the automated model-based synthesis of mediators at runtime. However, this approach has strong prerequisites. It requires the existence of adequate models of the systems to be connected. Many systems encountered in practice, on the other hand, do not come with models. In such cases models have to be constructed ex post (at runtime). Furthermore, adequate models must capture control as well as data aspects of a system. In most protocols, for instance, data parameters (e.g., session identifiers or sequence numbers) can influence system behavior. Models of such systems can be thought of as *interface programs*: Rather than covering only the control behavior, they describe explicitly which data values are relevant to the communication and have to be remembered and reused.

This thesis addresses the problem of inferring interface programs of systems at runtime using active automata learning techniques. Active automata learning uses a test-based and counterexample-driven approach to inferring models of black-box systems. The method has originally been introduced for finite automata (the popular L^* algorithm). Extending active learning to interface programs requires research in three directions: First, the efficiency of active learning algorithms has to be optimized to scale when dealing with data parameters. Second, techniques are needed for finding counterexamples driving the learning process in practice. Third, active learning has to be extended to richer models than Mealy machines or DFAs, capable of expressing interface programs.

The work presented in this thesis improves the state of the art in all three directions. More concretely, the contributions of this thesis are the following: first, an efficient active learning algorithm for DFAs and Mealy machines that combines the ideas of several known active learning algorithms in a non-trivial way; second, a framework for finding counterexamples in black-box scenarios, leveraging the incremental and monotonic evolution of hypothetical models characteristic of active automata learning; third, and most importantly, the technically involved extension of the partition/refinement-based approach of active learning to interface programs.

The impact of extending active learning to interface programs becomes apparent already for small systems. We inferred a simple data structure (a nested stack of overall capacity 16) as an interface program in no more than 20 seconds, using less than 45,000 tests and only 9 counterexamples. The corresponding Mealy machine model, on the other hand, would have more than 10^9 states already in the case of a very small finite data domain of size 4 and require significantly more than 10^9 tests when being inferred using the classic L^* algorithm.

Keywords: Interface Synthesis, Automata Learning, Regular Inference, Extended Finite State Machines, Register Automata, Register Mealy Machines.

List of papers

I **Introduction to Active Automata Learning from a Practical Perspective**
by Bernhard Steffen, Falk Howar, and Maik Merten. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6659:256-296.

II **From ZULU to RERS: Lessons Learned in the ZULU Challenge**
by Falk Howar, Bernhard Steffen, and Maik Merten. In Proceedings of the 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Part I, ISoLA 2010, *Lecture Notes in Computer Science*, *Springer Verlag*, 6415:687-704, 2010.

III **Automata Learning with Automated Alphabet Abstraction Refinement**
by Falk Howar, Bernhard Steffen, and Maik Merten. In Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6538:263-277, 2011.

IV **A Succinct Canonical Register Automaton Model**
by Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. In Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6996:366-380, 2011.

V **Inferring Canonical Register Automata**
by Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. In Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7148:251-266, 2012.

VI **Inferring Semantic Interfaces of Data Structures**
by Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Accepted for ISoLA 2012.

Comments on my participation

I **Introduction to Active Automata Learning from a Practical Perspective**
Most of the ideas and examples were worked out in discussions among the authors. I am co-author to all sections of the paper. I am lead-author of Section 5. I am one of two maintainers of LearnLib at the time and most of the strategies for handling counterexamples have been implemented by me.

II **From ZULU to RERS: Lessons Learned in the ZULU Challenge**
The approach to the ZULU competition presented in this paper evolved from discussions with Bernhard Steffen and Maik Merten. I am co-author to all sections of the paper. I am lead-author of Section 3. I carried out implementation and experiments.

III **Automata Learning with Automated Alphabet Abstraction Refinement**
The work presented in this paper is the continuation of work done in my Diploma thesis. The theoretical results were established in discussions with Bernhard Steffen and Maik Merten. I am co-author to all sections of the paper. I am lead-author of Section 4. I carried out implementation and experiments.

IV **A Succinct Canonical Register Automaton Model**
The canonical automaton model presented in this paper was worked out in discussions among the authors of this paper. I am co-author to all sections of the paper. I am the lead author of the comparison with other canonical automata.

V **Inferring Canonical Register Automata**
The algorithm for inferring register automata models was developed in discussions among the authors of the paper. I am co-author of the introduction and the lead-author for the rest of the paper. I carried out implementation and experiments.

VI **Inferring Semantic Interfaces of Data Structures**
The technical extensions of register automata learning to Register Mealy machines have been developed in discussions among the authors of the paper. I am co-author to all sections of the paper. The presentation in the paper is partly based on texts and figures from this thesis.

Other peer reviewed publications

Conference papers

- **Towards an Architecture for Runtime Interoperability**
  by Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Gang Huang, Nikolaos Georgantas, Paul Grace, Falk Howar, Paola Inverardi, Valérie Issarny, Massimo Paolucci, Animesh Pathak, Romina Spalazzese, Bernhard Steffen, and Bertrand Souville. In Proceedings of the 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Part II, ISoLA 2010, *Lecture Notes in Computer Science*, *Springer Verlag*, 6416:206-220, 2010.

- **Reusing System States by Active Learning Algorithms**
  by Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. In Proceedings of the 1st EternalS workshop, EternalS 2011, *Communications in Computer and Information Science*, 255:61-78, 2012.

- **On Handling Data in Automata Learning: Considerations from the CONNECT Perspective**
  by Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. In Proceedings of the 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Part II, ISoLA 2010, *Lecture Notes in Computer Science*, *Springer Verlag*, 6416:221-235, 2010.

- **Next Generation LearnLib**
  by Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. In Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6605:220-223, 2011.

- **Demonstrating Learning of Register Automata**
  by Maik Merten, Falk Howar, Bernhard Steffen, Sofia Cassel, and Bengt Jonsson. In Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, to appear 2012.

- **Inferring Automata with State-local Alphabet Abstractions**
  by Malte Isberner, Bernhard Steffen, and Falk Howar.
  Under Submission.

Book chapters

- **Practical Aspects Of Active Automata Learning**
  by Falk Howar, Maik Merten, Bernhard Steffen, and Tiziana Margaria. In Stefania Gnesi and Tiziana Margaria, editors, *Formal Methods for Industrial Critical Systems*, *Wiley-VCH*, to appear 2012.

Acknowledgements

First and foremost I want to thank Bernhard Steffen for guiding me through the past three years that led to this dissertation. Thank you for encouraging me, questioning me, supporting me, challenging me, and motivating me always in just the right moment of time.

I thank Bengt Jonsson and Sofia Cassel for a marvelous summer in Sweden, and for the great cooperation that led to and originated from this summer.

Then I want to thank Maik Merten, faithful companion in three years of sharing office, and traveling. Thank you for countless hours of inspiring discussions, all the fun time I had traveling (most of the time), and - of course - for providing the sound track to the past three years.

I am also grateful to everyone from the chair of Programming Systems for many work-related discussions and a great atmosphere.

Last but not least, I want to thank my wife Julia for all the patience and moral support without which I never would have survived the past three years.

Contents

List of Figures

1 Introduction

Software and the process of developing software has changed radically in the past two decades. Some 20 years ago most programs could be developed by single persons or small teams, implementing most of the functionality from scratch. With software aiding more and more processes and tasks in everyday life, systems began to grow and the need emerged to re-use functionality instead of developing everything from scratch in every application. Today almost any program uses third-party libraries and components. More importantly, systems in use today are no longer monolithic programs, they comprise a number of programs that interact with one another.

As an example consider any flight- or hotel-booking web-site. The application will be developed using a web framework, providing basic functionality, and it will use a number of libraries providing database abstraction, authentication, authorization, and other functionality common to these applications. Then, it will be integrated into a whole landscape of systems. For availability and prices of flights or hotels it will have to browse the contingents of carriers and travel agencies. Booking flights or making reservations in hotels, it has to interact with corresponding applications. In order to offer payment it has to be integrated with applications of payment providers, e.g., credit card companies, micro payment systems, etc. Finally, displaying additional information about travel locations, e.g., the current local weather, it has to obtain this information from some application.

With the growth of systems and the integration of systems into systems of systems, especially interoperability, i.e., the ability of different systems developed and maintained by different parties to work together, becomes a fundamental issue. In the past interoperability has mainly been addressed by standard-based middleware. Examples are CORBA [71], DCOM [83], or Web services [85]. Due to the diverse nature of distributed systems, however, one cannot hope for a single solution equally adequate in all scenarios. Integration of systems using different middleware technologies requires bridging between these technologies. While shared middleware eases the problem of interoperability, systems still have to be integrated manually at the level of application protocols. In the above example, the payment service might use a Web service protocol. In order to use it, the developers of the flight-booking service will still have to invoke it correctly, i.e., perform the right actions in the right order to complete payment transactions.

Ubiquitous computing and the internet of things aggravate the problem. Not only are we surrounded by an exploding number of computer systems, but these systems interact with one another. The flawless integration of these systems cannot be achieved at the time of development. The communication partners of these systems are simply not known by the time of development. In such scenarios interoperability gets a new quality. Not only do systems need to be compatible at the level of the used middleware technology. Encountering one another spontaneously, systems need to be compatible at the level of application protocols. Obviously, standardization cannot be a solution at the application level.

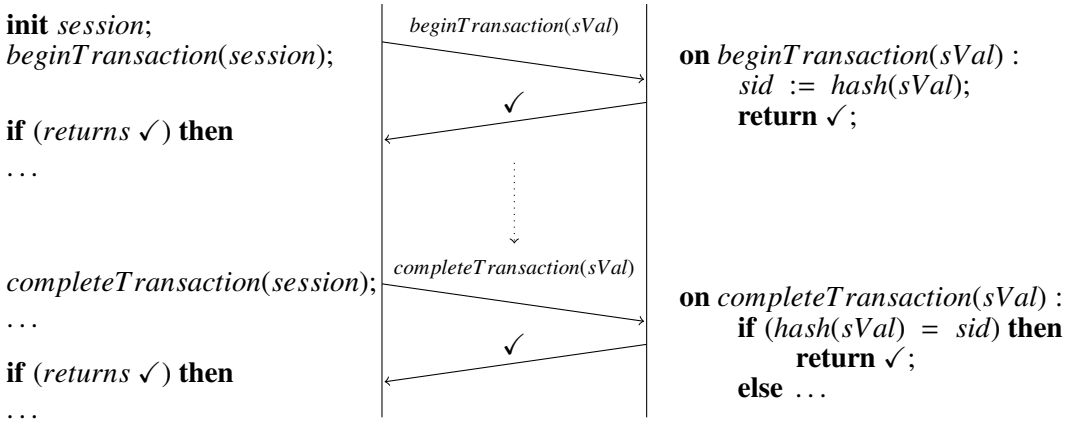

Figure 1.1: Integrated systems performing a payment transaction

With interoperability and integration of systems becoming more and more complex and more and more frequent at the same time, solutions are needed for automated integration of systems. One approach to this problem, taken by the CONNECT project [57] is the development of "Emergent Middleware", i.e., at set of enablers synthesizing mediators between systems automatically at runtime. The automated process envisioned in the CONNECT project has five main steps: (i) discovery of systems, (ii) inference of the (application) protocol of systems, (iii) model-based synthesis of mediators, (iv) functional and non-functional validation of mediators, and finally (v) deployment of mediators.

This thesis does focus on one particular step of the process: The automated inference of system behavior in the context of the envisioned automated integration of systems. One goal of the CONNECT project is the development of inference methods for models that can be used for synthesizing mediators based on active automata learning. Active automata learning uses a test-based approach for inferring models of black-box systems. Observations are made by actively interacting with a system under learning (SUL). The approach has been introduced in [7] for deterministic finite automata (DFAs). It has been extended to infer Mealy machine models in [70, 63], which are more adequate for describing reactive systems that produce outputs rather than accept or reject sequences of inputs. It has been used successfully to infer models of black-box systems in model-checking [72, 25], testing [52], and interface synthesis [4].

Active learning infers models of systems for which initially only a set of inputs is known by iterating two phases, namely

1. Test-based modeling: In this phase a hypothetical model is generated from observations made in tests.

2. Model-based testing: In this phase models generated in the first phase are tested for equivalence with the black-box system.

Active learning algorithms are formulated in a theoretical model that assumes a teacher, which answers two kinds of queries for the learning algorithm. So-called *membership queries* correspond to tests that have to be executed on a system under learning. The first phase of learning, test-based modeling, uses membership queries to construct a hypothetical

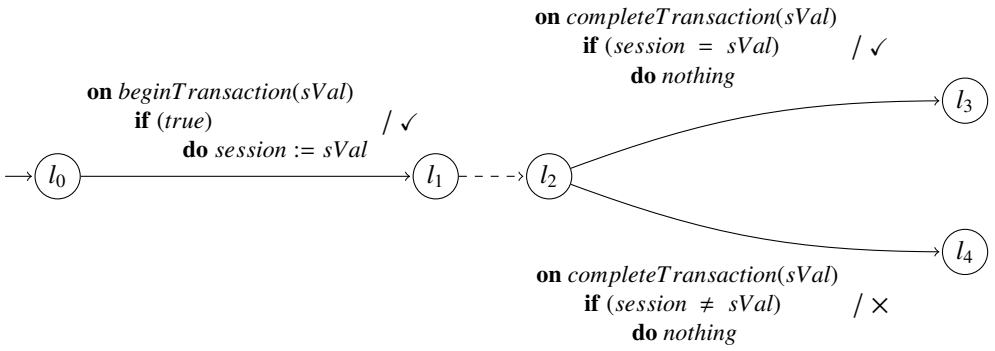

Figure 1.2: Informal model of a transaction protocol

model of the SUL. The second phase, model-based testing, is realized by so-called *equivalence queries*, testing whether a hypothetical model is equivalent to the SUL. If a model does not describe the behavior of a system faithfully, they provide a counterexample, i.e., a sequence of inputs proving the inequivalence of hypothesis and SUL.

Due to its test-based approach active learning can be considered the natural choice for inferring models of black-box systems, as envisioned in the CONNECT project. However, since it is restricted to finite automata, the inferred models are not sufficient as a basis for automated mediator synthesis as the following example shows.

Consider the payment service offered by a credit card company in the example of the flight-booking service. Payments are organized in transactions. When using the application, a certain protocol has to be followed. Transactions are identified by identifiers that have to be used in every step of the protocol. Figure 1.1 shows schematically the correct usage of such a service by an application: The application starts by creating an identifier and stores it in a variable named *session*. Then, the applications requests a new transaction using *session* in a message to the payment service. The payment service (in the right of the figure) stores a hash of the received identifier in a variable named *sid*, begins a new transaction associated with *sid* and sends a receipt to the application, denoted by ✓ in the figure. After providing the wiring details for the payment, which is omitted in the figure, the application requests the completion of the transaction in a message, using the identifier stored in *session*. Receiving this request, the payment service will check if the hash of the provided identifier matches the one stored in *sid*. If this is the case, the transaction will be completed successfully.

The behavior of the service does not only depend on the order in which messages are invoked. The data parameters that are used in the communication also influence on the services' behavior. To integrate the payment service successfully, it is necessary to use the primitives exposed by the service in the correct order, and to use the same identifier for the session in all invocations. An analysis of the requirements in the context of the CONNECT project is presented in [47].

Assuming that identifiers have to be unique, it is not an option to fix a single identifier in an application using the service. In order to achieve automatic integration of applications and services based on models of the systems behavior, it is necessary to express the requirements on the use of data parameters relevant to the communication in inferred models.

Figure 1.2 shows this idea informally. The displayed model of the above protocol has transitions, labeled by blocks of events, conditions, and actions. Comparing the labels with the pseudo-code in Figure 1.1, the events are methods of the payment service's application programming interface (API), i.e., entry points to the service. The conditions resemble the ones from the service. Finally, the assignments occur on both sides, in the application and in the service. However, in the figure the use of hashes is not visible and the name *session* is used for the variable. This emphasizes that the actions are not meant to encode *what* is actually happening in the service but rather *how* the application operating the service has to use data values from the communication.

The shown model is more than a behavioral model of the service. It is an *interface program* for the service. Rather than (only) covering the behavior of the service, it describes the protocol of an interaction with the service. Knowing the goal of the communication, e.g., a successfully completed transition (marked by output ✓ in the model), the model can be used to actively steer the communication with the service.

Interface programs describe explicitly how data values relevant to the communication have to be used. In order to make these requirements explicit in models, means are needed for expressing variables, conditions, and assignments. Thus, such models can be seen as (restricted) program structures capable of operating the interface of a system (hence, interface programs).

In order infer interface programs by means of active learning, active learning has to be extended beyond the realm of DFAs and Mealy machines. It has to be capable of inferring variables, conditions, and assignments describing the effect of data parameters used in the communication with a system.

1.1 Research problems addressed in this thesis

This section summarizes the research questions addressed in this thesis. The problem addressed in this thesis is

inferring interface programs for black-box systems
by means of test-based interaction with these systems.

Using active learning techniques to infer interface programs comes with a number of challenges. These challenges can be grouped into intrinsic and extrinsic ones. Intrinsic challenges are related to the method of active learning itself:

1. the vast amount of membership queries active learning requires,
2. the use of equivalence queries which do not exist in practice,
3. and (most importantly) that it is formulated for deterministic finite automata which do not suffice to express interface programs.

Extrinsic challenges are related to the application of active learning in concrete contexts, e.g., how to derive a set of inputs for a learning algorithm from a SUL, how to translate queries of a learning algorithm into actual invocations on a a SUL, or how to establish well-defined initial conditions for every query on a SUL.

One may be tempted to refer to the first category as conceptual problems, and to the second one as engineering ones. This is not correct, however. Both, intrinsic and extrinsic, problems have conceptual and engineering aspects and can be considered from both perspectives. E.g., the amount of membership queries may be reduced by using domain specific knowledge, which requires a conceptual idea for representing and exploiting such knowledge. To actually apply such a method in practice, however, for every domain a concrete solution has to be engineered. The problem of deriving a set of inputs from a SUL, on the other hand, is not only an engineering problem. It can be addressed at the conceptual level as well, e.g., by developing patterns or general methods for finding inputs for a SUL. Thus, rather than the problems, the solutions can be characterized as being conceptual or being engineered.

Using these classifications, the focus of this thesis is on conceptual aspects of the three intrinsic problems of active learning mentioned above. In more detail, the following problems are addressed.

Scalability of active learning algorithms: Active learning relies on test-based interaction with black-box systems. Learning performs huge numbers of tests. Most case studies report models with a couple of hundred states and small sets of inputs (e.g., [75, 52]), requiring already some hundred thousand tests to be executed on a SUL. Experiments on larger systems are rarely reported. To the knowledge of the author the largest reported model inferred through active learning is of a software router with 22,000 states and 7 inputs [74]. While this model exceeds typical case studies by more than one order of magnitude in size already, models of actual systems can still be larger by some orders of magnitude.

One common approach to this problem is the application of filters and domain-specific knowledge in order to compute (some) results of tests without actually testing the SUL [65, 64, 74]. The number of membership queries can also be reduced by a learning algorithm directly. As is shown in Section 2.2, the amount of membership queries used by a learning algorithm depends to a high degree on how counterexamples from equivalence queries are used by the algorithm. A learning algorithm is presented that can be combined with many of the strategies for handling counterexamples that have been suggested so far. Analyzing and categorizing these strategies in the context of the presented algorithm leads to a new effective strategy for handling counterexamples. Basis for this analysis is an observation about the structure of counterexamples, presented in [82] (Paper I).

A second approach to large models is the inference of partial models of systems, usually related to some property to be verified on the system (e.g., [35]). Inferring partial models requires a means for dealing with partial models in the learning algorithm as well as replacing equivalence of models by a relation that supports partiality of an inferred model. In Section 3.1 an idea for inferring only certain aspects of a SUL is presented.

Testing equivalence of black-box systems: Active learning relies on equivalence queries, i.e., tests for equivalence of a black-box system and an inferred model of this system. While equivalence queries provide a theoretical framework for formulating active learning algorithms, they do not exist in practice. By definition, a model of

an unknown black-box system cannot be verified by a finite number of tests. It can only be falsified. Using active learning in practice, equivalence queries can only be approximated by means of testing. This has the consequence that regular inference in practice is neither sound nor complete.

One important step towards making active learning a valuable tool for inference of models of black-box systems is developing means for approximating equivalence queries in practice. Due to the conceptual proximity of active learning and conformance testing, which is discussed in [12], methods from conformance testing have been suggested as a natural match. These methods aim at proving equivalence (under certain additional assumptions) by means of single tests, which is very expensive. In the course of learning, however, the inferred model is only tested successfully for equivalence with the black-box once, at the very end of the learning experiment. The vast majority of equivalence queries are be unsuccessful and return a counterexample. Thus, in most cases what really is needed is a method for finding counterexamples efficiently.

A second problem of equivalence queries is that they are not incremental. While subsequent hypothesis model are strongly related, equivalence queries test each hypothesis independently. In practice this leads to high costs in terms of tests to be executed on a SUL. Section 3.2 presents a strategy for realizing equivalence queries incrementally, focusing on finding counterexamples rather than on testing equivalence unsuccessfully for all but one hypothesis model. This strategy has been evaluated in the ZULU challenge [26], a contest in active automata learning with membership queries, where it was an integral part of the winning algorithm in the competition. Detailed results of the evaluation are discussed in [50] (Paper II).

Active learning for interface programs: Active learning produces Mealy machine models or DFAs. While these models are often sufficient for organizing tests or for verification, where one can encode data parameters statically in a set of predefined inputs covering (potentially) interesting cases, they cannot be used as a basis for proper integration of systems at runtime. In order to achieve integration of systems based on models, interface programs are needed, modeling explicitly the requirements on the use of data values in the communication. This changes the quality of models. Rather than being pure descriptions of sets of program traces, they become (restricted) programs capable of operating the interface of a system actively, storing values from the communication and re-using these values at a later point in time. In order to use active learning successfully in contexts as the CONNECT project [57], it has to be extended to produce such richer models. Two problems have to be addressed in particular.

1. Active learning has to be capable of dealing with infinite sets of inputs comprising data values from unbounded domains (e.g., natural numbers or strings).
2. Active learning has to be extended to infer models that make explicit the aforementioned requirements on data values used in the interaction with a system.

One common approach to the first problem, i.e., inferring models of systems with infinite sets of inputs, is the use of abstractions on the set of inputs. Such abstractions are commonly realized by placing a mapper in between the SUL and the learning

algorithm [58]. The learning algorithm then works on a set of abstract inputs, which the mapper translates into concrete inputs to be executed on the SUL. However, the construction of such mappers is a laborious and error-prone task that has to be executed manually. It especially involves finding sound abstractions prior to learning, i.e., ones that do not introduce non-determinism. This usually leads to a number of iterations of failed learning experiments (due to observed non-determinism) and manual refinements to the mapper.

In Section 4.1, which briefly introduces the ideas from [51] (Paper III), it is discussed how the partition-refinement based approach of active learning can be extended to the set of inputs in order to derive a sound and optimal abstraction automatically in the course of learning. A main feature of this approach is that it does not need a mapper as the learning algorithm works directly on concrete representative inputs.

The second problem, i.e., inferring interface programs for real systems capable of storing data values and using these in the subsequent communication, is conceptually more challenging. While the use of abstraction is still based on active learning of finite automata, for this second problem active learning itself has to be extended to richer models.

Concretely, a means of expressing storage of data values and operations on these data values is required. Imagine, e.g., a session identifier or a sequence number, which influences the behavior of a system. In order to describe such phenomena in finite models, they have to be made explicit. This requires the development of a new automaton model comprising registers, storing data values, and transitions that can operate on these registers. Also, adequate notions of regularity and canonicity have to be established. This is sketched in Section 4.2 and presented in [23] (Paper IV).

Then, in order to infer interface programs, an active learning algorithm capable of dealing with models comprising conditions and assignments is needed. Extending active learning to the extended automaton model requires the adaption of the key ideas of active learning. Active learning relies heavily on the well-known Nerode-relation [69] and on the characterization of states of a SUL, which it provides by means of suffixes. When inferring models with registers that store data, especially this suffix-based characterization of states has to be extended. The necessary steps are sketched in Section 4.3. The resulting active learning algorithm is presented in [49] (Paper V).

Finally, black-box components usually can be modeled as reactive systems, providing output (with data values) on inputs. Corresponding to using Mealy machines instead of deterministic finite automata in classic active learning, it is possible to model outputs with data values directly. While, as in the classic case, this does not change the expressivity of the obtained models, it leads to a dramatically increased performance of learning algorithms. A Mealy machine variant of systems storing data values and its effect on the performance of learning algorithms is discussed in Section 4.4, summarizing [46] (Paper VI).

Key to progress in all research problems addressed in this thesis is rethinking active learning and identifying and leveraging the conceptual building blocks of active learning. These are

- the Nerode-relation and Myhill/Nerode-theorem [69], and
- the partition-refinement based approach to incremental construction of models driven by counterexamples.

All research problems discussed above relate to these building blocks in some respect: Reducing the number of membership queries requires analyzing how the Nerode-relation can be realized optimally by a minimal set of suffixes. Key to inferring partial models and to incremental equivalence queries is the breath-first style of exploration of the state space that can be enforced by the analysis of counterexamples. Alphabet abstraction refinement incorporates the partition-refinement-based approach to the on-the-fly construction of sets of inputs, using an equivalence relation on an infinite set of inputs that strongly resembles the Nerode-relation. Finally, extending active learning to register automata models, requires an extended Nerode-relation, allowing for the application of a multi-dimensional partition-refinement-based approach (on states, registers, and inputs) for inferring such models.

1.2 Organization

The thesis is organized as follows. In the next chapter, an introduction to inferring Mealy machine models is given. Presentation is based on the observation pack algorithm, a frame algorithm that can be instantiated with different strategies for handling counterexamples. The chapter presents original results, exceeding the results of the corresponding papers. The presented observation pack algorithm is due to the author and has not been presented in detail before, as is the analysis of different strategies for handling counterexamples. Results from applying both ideas in the context of the ZULU competition are discussed in [50] (Paper II), which, however, covers neither the technical details of the observation pack algorithm, nor does it include an analysis of different approaches of handling counterexamples. This analysis is based on ideas presented in [82] (Paper I) but clearly exceeds these.

In Chapter 3, two ideas are discussed that have in common a “local” approach to inference. First it is shown, how local exploration of the state space allows for learning partial models of big systems. Then, it is investigated how the (local) differences in subsequent hypothesis models obtained during inference can be understood as modifications to one evolving hypothesis model, which leads to a new incremental version of equivalence queries. While the positive effect of using incremental equivalence in practice is discussed in [50] (Paper II), the “local” perspective to active learning discussed this chapter is original.

Chapter 4 sketches the essential ideas for inferring systems with data, i.e., with infinite sets of inputs or even infinite state spaces. In particular, a new automaton model with a corresponding language model, and Nerode-relation are presented. These are then used as basis for an active learning algorithm. The chapter provides introductions to the corresponding papers [51] (Paper III), [23] (Paper IV), [49] (Paper V), and [46] (Paper VI) along with examples and intuitions. The papers contain detailed technical accounts of the presented results.

Related work is discussed in Chapter 5 before concluding in Chapter 6.

2 Inferring Mealy machines

Active automata learning, also sometimes reffered to as regular inference, aims at inferring automata models of black-box systems for which initially only a set of inputs is known. Angluin's popular L^* algorithm [7] infers an unknown regular language $\mathcal{L}$, producing a minimal deterministic finite acceptor (DFA) for $\mathcal{L}$. It has been adapted to Mealy machines by Niese [70]. This chapter presents the *observation pack* algorithm, a frame algorithm consolidating the central conceptual ideas in regular inference. It can be concretized to work for regular languages as well as for Mealy machines. Presentation is be based on Mealy machines to allow for inclusion of results on handling counterexamples from [82] (Paper I).

2.1 Mealy machines

In this section the formal notation introduced in [82] (Paper I) for Mealy machines and their sets of traces is briefly recited. For one thing, the notation will serve as a formal basis for the subsequent sections. More importantly, however, this allows for revisiting the central ideas of the Nerode-relation and the corresponding Myhill/Nerode-theorem for regular languages [69] while discussing their adaption to the Mealy scenario.

Definition 1 (Mealy machine) *A Mealy machine is a tuple* $\mathcal{M} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function*.* □

A Mealy machine processes sequences of inputs (input words or simply words), producing outputs. Let therefore ε denote the empty word, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ be the set of all words of length greater than zero. Then, let $[\![\mathcal{M}]\!] : \Sigma^+ \to \Omega$ be the *semantic functional* of $\mathcal{M}$. In order to describe $[\![\mathcal{M}]\!]$ in terms of $\mathcal{M}$, we extend the transition function to $\delta^* : Q \times \Sigma^* \to Q$ by defining inductively $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ for $q \in Q$ and $aw \in \Sigma^+$ with $w \in \Sigma^*$. Let $[\![\mathcal{M}]\!]$ now simply be defined to be the last output of $\mathcal{M}$ on a particular word, i.e.,

$$[\![\mathcal{M}]\!](wa) \;=_{def}\; \lambda(\, \delta^*(q_0, w),\; a) \qquad \text{for } wa \in \Sigma^+.$$

Figure 2.1 shows a graphical representation of a Mealy machine for a 3-place buffer, which will be used as an example throughout the thesis. The buffer has four states, an initial (empty) state, and one state per number of potential elements in the buffer. The input **put**

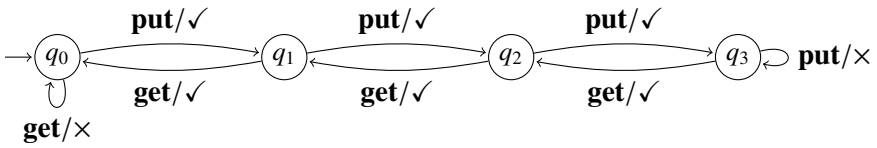

Figure 2.1: Mealy machine model of 3-place buffer

adds one element to the buffer, and the input **get** removes one element from the buffer. Almost all operations produce output ✓. Only unsuccessful operations produce output ×.

To describe precisely the class of systems that can be modeled as Mealy machines and of which Mealy machine models can be inferred, it is necessary to characterize the class of mappings $T : \Sigma^+ \to \Omega$ that can be understood as the semantic functional of some Mealy machine. Key to this characterization is the equivalence induced by T on input words.

Definition 2 (Equivalence wrt. T) *Two words $u, u' \in \Sigma^*$ are equivalent wrt. $\equiv_T$, denoted by $u \equiv_T u'$, iff*

$$\forall v \in \Sigma^+ \,.\, T(uv) \;=\; T(u'v). \qquad \square$$

This equivalence corresponds to the well-known Nerode relation for regular languages [69]. It can be reformulated in terms of residual languages [28], or in the case of Mealy machines in terms of *residual functionals*. Let $u^{-1}T : \Sigma^+ \to \Omega$ with $u^{-1}T(v) =_{def} T(uv)$ for $u \in \Sigma^*$ and $v \in \Sigma^+$. Let $[u]$ be the equivalence class of u wrt. $\equiv_T$, i.e., the set of words with identical residual functional.

Since Mealy machines have finitely many states, only mappings $T : \Sigma^+ \to \Omega$ where $\equiv_T$ has finite index can be represented as Mealy machines. On the other hand, the semantic functional of a Mealy machine can have only finitely many *residual functionals* since the Mealy machine has only a finite number of states. This directly yields the following equivalent to the Myhill/Nerode-theorem for regular languages [69].

Theorem 1 (Characterization theorem) *A mapping $T : \Sigma^+ \to \Omega$ is a semantic functional for some Mealy machine iff $\equiv_T$ has finite index.* $\square$

A proof of the theorem is presented in [82] (Paper I). In reminiscence of the classification of regular languages, let *regular mappings* be mappings $T : \Sigma^+ \to \Omega$ that can be represented as Mealy machines. From a regular mapping T one can straightforwardly construct the Mealy machine representation $\mathcal{M}_T$ of T: Let $\mathcal{M}_T = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ such that

- The set of states is defined by the classes of $\equiv_T$: one state $q_{[u]}$ for every class $[u]$ of $\equiv_T$.
- The initial state is the state for the class of the empty word, i.e., $q_0 = q_{[\epsilon]}$.
- The sets Σ and Ω are determined using the domain and the range of T.
- The transition function is defined as $\delta(q_{[u]}, a) =_{def} q_{[ua]}$ for $a \in \Sigma$.
- The output function is defined as $\lambda(q_{[u]}, a) =_{def} T(ua)$ for $a \in \Sigma$.

The finite index of $\equiv_T$ guarantees that $\mathcal{M}_T$ is well-defined. Thus, $\mathcal{M}_T$ is the *canonical* Mealy machine for T.

This approach to automata construction is a central element in active learning. Every learning algorithm will use a variant of this method for constructing models. The construction has to be slightly modified to work on finite sets of words rather than on the equivalence classes of $\equiv_T$. Also, these equivalence classes have to be characterized by finite subsets of their residuals.

2.2 Active learning for Mealy machines

Assume a system under learning with a a set of inputs Σ and a set of outputs Ω that can be represented as a Mealy machine. Active automata learning is based on two main aspects of Theorem 1. It can be summarized as

1. finding a finite set $U \subset \Sigma^*$ of *prefixes* that contains a word from every class of the (unknown) equivalence relation $\equiv_{SUL}$, and

2. finding a finite set $V \subset \Sigma^+$ of *suffixes* that is sufficient to realize the Nerode-relation on U, i.e., such that $u \not\equiv_{SUL} u'$ implies $[\![SUL]\!](uv) \neq [\![SUL]\!](u'v)$ for $u, u' \in U$ and some $v \in V$.

Certainly, it is possible to find a finite set of prefixes since we assume $[\![SUL]\!]$ to be regular. The existence of a finite set of suffixes follows directly from Definition 2. For any two classes of $\equiv_{SUL}$ there is at least one *distinguishing suffix*, and every distinguishing suffix partitions the set of classes of $\equiv_{SUL}$ in at least two blocks. Hence, it is possible to find a set V of suffixes that is bounded in size by the index of $\equiv_{SUL}$.

Active learning algorithms are formulated in the MAT-learning model [7], which assumes the existence of a minimally adequate teacher (MAT), answering two kinds of queries.

Membership queries test for the output of a word $w \in \Sigma^+$. Sometimes $MQ(w)$ will be used instead of $[\![SUL]\!](w)$ to emphasize that the value has to be determined by a test on the SUL.

Equivalence queries test whether an intermediate hypothesis automaton $\mathcal{H}$ is equivalent to the SUL, i.e., if $[\![\mathcal{H}]\!] = [\![SUL]\!]$. In case that hypothesis and system are not equivalent, an equivalence query delivers a *counterexample*, i.e., a word $w \in \Sigma^+$ for which $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$. Equivalence queries will be denoted by $EQ(\mathcal{H})$ in pseudo-code listings.

Corresponding to the two kinds of queries, inference is organized in two phases, alternated iteratively. In the first phase a hypothesis model is derived and iteratively refined using membership queries. In the second phase hypothesis models are verified by means of equivalence queries.

This thesis presents a new active learning algorithm for the MAT model: The observation pack algorithm infers models of back-box systems by means of membership queries and equivalence queries. Technically, the algorithm combines the idea of using a discrimination tree for learning regular languages [60] with a localized version of the observation tables

of [7]. The name "observation pack" is chosen in reminiscence of the abstract concept of an observation pack discussed in [8]. Please note, however, that in [8] no concrete learning algorithm is presented. The algorithm presented here is due to the author.

2.2.1 Hypothesis construction

Let us begin by describing how membership queries can be used to construct hypothesis models. In particular this requires data structures for maintaining the sets of prefixes and suffixes discussed above.

Definition 3 (Component) *A component C is a tuple $\langle U, u_0, V, T\rangle$ where*

- $U \subset \Sigma^*$ *is a finite set of* prefixes*,*
- $u_0 \in U$ *is the unique* access sequence *of the component,*
- $V \subset \Sigma^*$ *is an finite ordered set of* suffixes $v_1, \ldots, v_k$*, and*
- $T : U \times V \to \Omega$ *is the* observation table*.*

Let $\mathbf{row}(u)$ *with $u \in U$ be the sequence $\langle T(u, v_1), \ldots, T(u, v_k)\rangle$ for $k = |V|$, i.e., the "row" of u in T.* □

A component consists of a set of prefixes, a dedicated prefix, i.e., its *access sequence*, a set of suffixes, and a mapping from pairs of prefixes and suffixes to outputs, approximating the Nerode-relation on the set of prefixes by the set of suffixes. The mapping T is derived by means of membership queries. In a hypothesis automaton, each component will correspond to one state. The prefixes of a component represent the transitions that end in this state. In order to represent different states by different components, (new) prefixes have to be related to components.

Definition 4 (Discrimination tree) *A discrimination tree $\mathcal{T}$ is a rooted tree defined by the tuple $\langle N, n_0, E, \lambda, L\rangle$ where*

- N *is a finite set of* nodes*,*
- $n_0 \in N$ *is the* root *of the tree,*
- $E \subseteq N \times N$ *is the finite set of* edges*,*
- $\lambda = \lambda_E \cup \lambda_N$ assigns labels *to edges and nodes, where*
  $\lambda_E : E \to \Omega$*,*
  $\lambda_N : N \to \Sigma^*$*, and*
- $L \subseteq N$ *is the set of* leaves*.*

For $n \in N$, let $\mathbf{parent}(n) = \{n' \in N \mid (n', n) \in E\}$*. Since $\mathcal{T}$ is a rooted tree, we have* $|\mathbf{parent}(n_0)| = 0$*, and* $|\mathbf{parent}(n)| = 1$ *for $n \in N \setminus \{n_0\}$. Let thus* $\mathbf{parent}(n)$ *also denote the unique predecessor of n. A node is a leaf if it is not the predecessor of some node.* □

A discrimination tree is used to sort new words into a set of components (described below). Inner nodes of this tree are labeled with suffixes; edges in the tree are labeled with outputs. Leaves of the tree are labeled by prefixes. For every leaf in the tree there is a corresponding component in the set of components, the label of the leaf corresponding to the access sequence of the component.

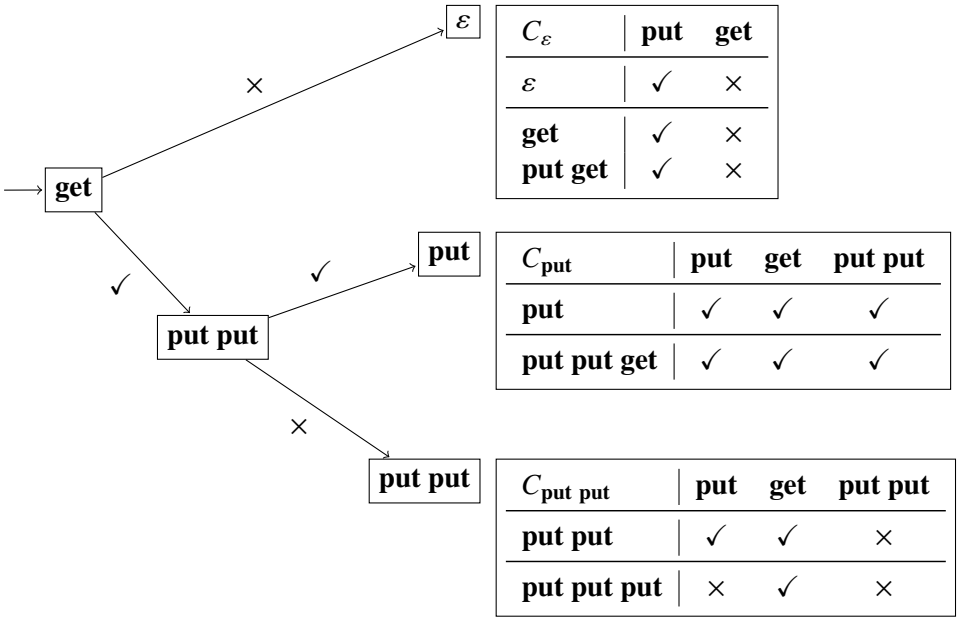

Figure 2.2: Unclosed observation pack for the 3-place buffer from Figure 2.1. The bottom-most component is unclosed.

Definition 5 (Observation Pack) *An observation pack P is a tuple $\langle \mathcal{T}, C \rangle$ where $\mathcal{T}$ is a discrimination tree and C is a set of components.* □

The set of access sequences in an observation pack, denoted by $U_0(C)$, is the set of access sequences of its components. The set of prefixes is the union of the (disjoint) sets of prefixes of all components. The set of suffixes of an observation pack is the union of the sets of suffixes of all its components.

Figure 2.2 shows an intermediate observation pack for the 3-place buffer from Figure 2.1. The discrimination tree is shown on the left-hand side, the components are shown on the right-hand side of the figure. The inner nodes of the discrimination tree are labeled by suffixes **get** and **put put**. Every leaf in the tree corresponds to one component. The labels of the leaves equal the access sequences of the components. The set of access sequences is prefix-closed and represents a subset of the states of the 3-place buffer (there is no component for the state representing three elements in the buffer, yet).

Initially an observation pack consists only of one component $\langle U, u_0, V, T \rangle$ for the empty prefix, i.e., $U = \{\varepsilon\}$, and $u_0 = \varepsilon$. The set of suffixes V is initialized as Σ in the case of inferring Mealy machine models. The discrimination tree is initialized with only one leaf labeled by ε, which also is the root of the tree.

A new word u is *sifted* into the observation pack (Algorithm 1), by sinking it into the discrimination tree. At each inner node on the path from the root a membership query is performed for the concatenation of u and the suffix v labeling the node (line 4). The outcome of the membership query determines which edge to follow (line 6). New words are added as prefixes to the component that corresponds to the leaf reached when sifting the word into the tree (lines 15-16).

Algorithm 1 Sift($\mathcal{T}$,C,u)

Input: A discrimination tree $\mathcal{T} = \langle N, n_0, E, \lambda, L\rangle$, a set of components C, and a new prefix $u \in \Sigma^*$

Output: A new component or 'OK'

```
1:  n := n_0                                          ▷ Start at root node
2:  while n ∉ L do                                    ▷ While current node is leaf ...
3:      v := λ(n)                                     ▷ Get suffix for node
4:      o := MQ(uv)                                   ▷ Check suffix on prefix
5:      if exists e = (n, n') in E with λ(e) = o then ▷ Determine successor
6:          n := n'
7:      else                                          ▷ No successor: new leaf
8:          Create new node n_u
9:          N := N ∪ {n_u},    E := E ∪ {(n, n_u)}    ▷ Update tree
10:         λ(n_u) := u,    λ((n, n_u)) := o          ▷ Update labels
11:         Create component C_u                      ▷ Create new component
12:         return C_u                                ▷ return component
13:     end if
14: end while
15: u' := λ(n)                                        ▷ Get component for leaf
16: add u to C_u' (for C_u' ∈ C)                      ▷ Add prefix to component
17: return 'OK'                                       ▷ Done
```

In case, at some inner node no appropriate edge is found, a new leaf is created and connected to the inner node by an edge, labeled with the outcome of the corresponding membership query. A new component will be created for the new leaf, using u as access sequence of the new component (lines 8-11). The set of suffixes of a new component will be initialized using the suffixes on the path in the tree leading to the newly created leaf, or as the set of suffixes of the observation pack, depending on how counterexamples are handled. This is discussed in detail in the next section.

In order to be able to construct well-defined hypothesis automata from an observation pack, one condition has to hold on the components. A component is closed if all prefixes are equivalent wrt. to the approximated Nerode-relation, i.e., if the rows of all prefixes in T are identical.

Definition 6 (Closedness) *A component $\langle U, u_0, V, T\rangle$ is* closed *if $row(u) = row(u_0)$ for all $u \in U$.* □

An observation pack is *closed* if all components are closed. Uncloseness of a component means that two prefixes u_0 and u in the component are not equivalent wrt. the set of suffixes. In particular, there is one suffix in V such that $T(u_0, v) \neq T(u, v)$. This means that $u \not\equiv_T u_0$. Since one component represents one state, and the two prefixes do not lead to the same state in the canonical Mealy machine for T, the prefixes have to be organized into different components. In the example in Figure 2.2 the bottom-most component is unclosed: The words **put put** and **put put put** do not lead to the same state in the 3-place buffer, which is proven by the suffix **put**.

In such a case the component will be split as shown in Algorithm 2. The prefix u will become the access sequence of the new component. All words u' in the old component will

Algorithm 2 Split($\mathcal{T}$,C,C,u,v)

Input: A discrimination tree $\mathcal{T} = \langle N, n_0, E, \lambda, L\rangle$, a set of components C, a dedicated component $C = \langle U, u_0, V, T\rangle$ from C, a prefix $u \in U$, and a suffix $v \in V$ s.t. $T(u_0, v) \neq T(u, v)$

Output: A new component

1: Create component $C_u = \langle \emptyset, u, V, \emptyset\rangle$ in C
2: **for** $u' \in U$ **do** ▷ Transfer prefixes
3: **if** $T(u_0, v) \neq T(u', v)$ **then**
4: transfer u' from C to C_u
5: **end if**
6: **end for**
7: Let $n \in L$ where $\lambda(n) = u_0$ ▷ Select old leaf for u
8: Create new node n_u
9: Create new inner node n_v
10: $N := N \cup \{n_u, n_v\}$ ▷ Add new nodes to tree
11: $\lambda(n_v) := v$ ▷ Add new suffix to labels
12: **if** $n = n_0$ **then** ▷ Original node was root?
13: $n_0 := n_v$ ▷ Then: replace root
14: **else** ▷ Otherwise: replace old leaf by new inner node
15: $n_p :=$ **parent**(n) ▷ Get old parent (singleton set)
16: $l := \lambda((n_p, n))$ ▷ Get old edge label
17: $E := (E \setminus \{(n_p, n)\}) \cup \{(n_p, n_v)\}$ ▷ Update edges
18: $\lambda((n_p, n_v)) := l$ ▷ Update labels
19: **end if**
20: $E := E \cup \{(n_v, n), (n_v, n_u)\}$ ▷ Add new edges to leaves
21: $\lambda((n_v, n)) := T(u_0, v), \quad \lambda((n_v, n_u)) := T(u, v)$ ▷ Update labels
22: **return** C_u ▷ Done

be sorted into one of the components, depending on $T(u', v)$ (lines 1-6). The new component uses the same set of suffixes as the old component. Hence, no membership queries are spent splitting a component.

After splitting the component, the discrimination tree has to be updated accordingly. The leaf representing the old component has to be split into two leaves, one for the old and for the new component. A new inner node labeled by v has to be introduced, replacing the old leaf in the tree. The two leaves will be connected to this new inner node (lines 7-21).

This results in the procedure shown in Algorithm 3. First, components are completed by performing membership queries in order to complete the table mappings (line 3). Then a check for unclosed components is performed. These are split, and newly created components are added to a work list (lines 4-6). Then, until the work list is empty, one-letter continuations of access sequences of components in the work list are added to the observation pack by sifting them into the discrimination tree. This, of course, may lead to further new components (lines 7-15). Once the work list is empty, the algorithm starts over by completing components. This is done until each component of the pack is complete and closed. From a complete and closed observation pack $\langle \mathcal{T}, C\rangle$ a hypothesis $\mathcal{H} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda\rangle$ can be constructed (line 17) following the approach from Section 2.1:

- The set of states is defined by the access sequences: one state q_u for every prefix u in the set of access sequences $U_0(C)$.

Algorithm 3 ClosePack($\mathcal{T}$,C)

Input: An observation pack $\langle \mathcal{T}, C \rangle$
Output: Hypothesis $\mathcal{H}$

```
1:  W := ∅        (only in first iteration: W := {C_ε})        ▷ Init. list of new components
2:  while unclosed or incomplete do                            ▷ Loop until stable
3:      complete components                                    ▷ Perform membership queries
4:      Let C = ⟨U, u_0, V, T⟩ with u ∈ U, v ∈ V s.t. T(u_0, v) ≠ T(u, v)   ▷ Select unclosed
5:      C_u := Split(T, C, C, u, v)                            ▷ Split component
6:      W := W ∪ {C_u}                                         ▷ Add new component to W
7:      while W ≠ ∅ do                                         ▷ Process new components
8:          C_u := poll(W)                                     ▷ Remove new component from W
9:          for a ∈ Σ do
10:             c := Sift(T, C, ua)                            ▷ Sift new prefixes to components
11:             if c ≠ 'OK' then                               ▷ Sifted to new component?
12:                 W := W ∪ {c}                               ▷ Then: add new component to W
13:             end if
14:         end for
15:     end while
16: end while
17: construct H from ⟨T, C⟩                                    ▷ As described in Section 2.2.1
18: return H                                                   ▷ Done
```

- The initial state is the state for the empty word ε.

- The sets Σ and Ω are known as inputs to the learning algorithm.

- The transition function is defined as $\delta(q_u, a) =_{def} q_{u'}$ if ua is in the component with access sequence u'.

- The output function is defined as $\lambda(q_u, a) =_{def} T(u, a)$, where T is the table mapping of component C_u.

The closedness of the observation pack and the initialization of all sets of suffixes as supersets of Σ guarantee that $\mathcal{H}$ is well-defined. While the former ensures that every transition has a defined destination, the latter guarantees that the output function can be defined from the table mappings. Finally, the initial state exists since ε is an access sequence. By construction, $[\![\mathcal{H}]\!]$ equals $[\![SUL]\!]$ for all prefixes in the observation pack.

2.2.2 Hypothesis verification

Once a hypothesis $\mathcal{H}$ is produced from the observation pack, an equivalence query can be used to find a counterexample, i.e., an input word $w \in \Sigma^+$ for which $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$. In the literature different methods are discussed for using counterexamples in active learning. These methods fall into two categories.

1. Suffix-based methods for handling counterexamples add suffixes of a counterexample to the set of distinguishing suffixes, resulting in unclosedness of the observations directly.

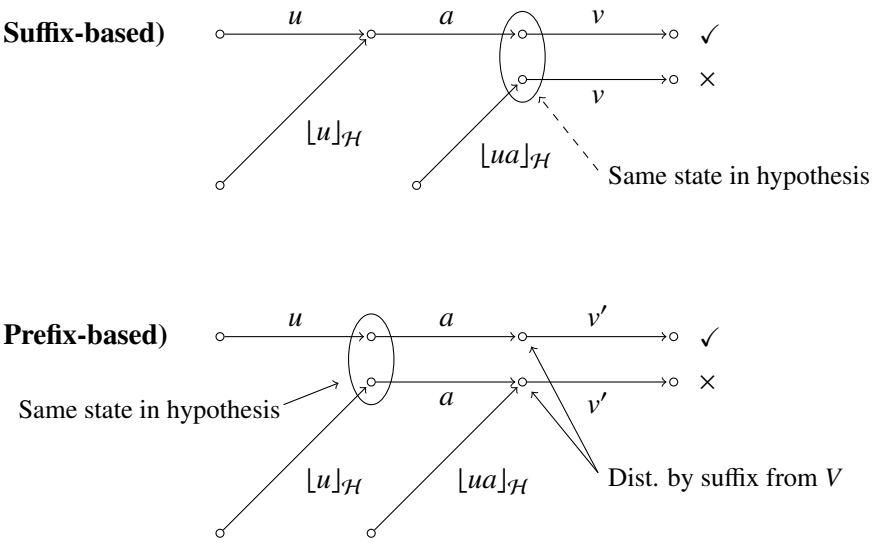

Figure 2.3: Exploiting counterexamples: suffix-based vs. prefix-based strategies

2. Prefix-based methods add prefixes of a counterexample to the set of prefixes used by the learning algorithm [7, 60], resulting in *inconsistency* of the observations, i.e., observations that would lead to non-deterministic hypothesis automata. Inconsistencies are handled by step-wisely extending the set of suffixes with suffixes resolving non-determinism. These suffixes in general are not equal to suffixes of a counterexample.

Methods of both categories can be further classified using the following fundamental theorem about counterexamples from [82] (Paper I). Variants of this theorem are used in subsequent chapters of this thesis. Let $\lfloor u \rfloor_{\mathcal{H}}$ denote the access sequence of the component corresponding to the state reached by u in $\mathcal{H}$.

Theorem 2 (Counterexample Decomposition) *For every counterexample $w \in \Sigma^+$ (i.e., $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$) there exists a decomposition $w = uav$ into an access sequence $u \in U_0(C)$, an input $a \in \Sigma$, and a suffix $v \in \Sigma^+$ such that $[\![SUL]\!](uav) \neq [\![SUL]\!](\lfloor ua \rfloor_{\mathcal{H}} v)$.* □

The theorem is the basis for two observations about counterexamples.

First, it states that at least one prefix of the counterexample does not lead to the same state in the SUL as its access sequence (from the hypothesis). For some suffix v' from the set of distinguishing suffixes it holds that $[\![SUL]\!](uav') \neq [\![SUL]\!](\lfloor ua \rfloor_{\mathcal{H}} v')$. In the extreme case $v = v'$ and both are of length one. It follows that u and $\lfloor u \rfloor_{\mathcal{H}}$, leading to the same state in the hypothesis, can be distinguished by av'. The situation is shown schematically in the lower half of Figure 2.3. Prefix-based strategies for handling rely on this perspective.

Second, it also follows from the theorem that at least one suffix of a counterexample will lead to unclosedness: There is a unique decomposition uav with a shortest prefix u that enjoys the properties stated in the theorem. For this decomposition $[\![SUL]\!](uav) = [\![SUL]\!](\lfloor u \rfloor_{\mathcal{H}} av)$. It follows transitively that $[\![SUL]\!](\lfloor u \rfloor_{\mathcal{H}} av) \neq [\![SUL]\!](\lfloor ua \rfloor_{\mathcal{H}} v)$. Since the words $\lfloor u \rfloor_{\mathcal{H}} a$ and $\lfloor ua \rfloor_{\mathcal{H}}$ are both in the set of prefixes (even in the same component!), the suffix v will lead to unclosedness. The situation is shown schematically in the upper half of Figure 2.3. This perspective is the backbone for suffix-based strategies for handling counterexamples.

Strategies for handling counterexamples can now be characterized in two additional respects. First, by the number of suffixes (or prefixes, respectively) that are used from counterexamples. Second, by the prefixes to which the suffixes are applied. Traditionally, suffixes have been used uniformly for all prefixes. Only Kearns and Vazirani present an approach for using suffixes to distinguish prefixes only pairwisely in [60]. Differing (1) in whether all or only some suffixes (prefixes) of a counterexample are used and (2) in whether these suffixes are applied to all or only some prefixes, the following strategies for exploiting counterexamples have been suggested.

Suffix-based methods. Since, as discussed above, the prefixes $\lfloor ua \rfloor_{\mathcal{H}}$ and $\lfloor u \rfloor_{\mathcal{H}} a$ are in the same component of the observation pack, it suffices to add all suffixes of a counterexample to all sets of suffixes in order to produce unclosedness in one of the components, which in turn will lead to splitting the corresponding component, and yield a new state in the next hypothesis. This strategy will be referred to as **AllGlobally**. It has been suggested by Maler and Pnueli in [62] for inferring DFAs. Optimizations to this strategy, using only a suffix-closed subset of the suffixes and adding these suffixes incrementally are presented in [77] and [54] for Mealy machines.

As shown in [75] for DFAs and in [82] (Paper I) for Mealy machines, a concrete decomposition of a counterexample can be determined using a binary search over the counterexample. This search provides one suffix that subsequently will lead to unclosedness in one of the components. Adding only this suffix to all sets of suffixes, referred to as **OneGlobally**, thus is sufficient to guarantee progress.

Finally, the decomposition of a counterexample also provides the short prefix of the exact component to split. Adding one suffix to this component, using a binary search to determine the decomposition of the counterexample, will be referred to as **OneLocally**. This strategy is due to the author and has been presented and applied first in the context of the ZULU competition [50] (Paper II).

The fourth possible strategy, i.e., adding all suffixes of a counterexample to one component has not been investigated yet.

Prefix-based methods. The second category of strategies for handling counterexamples does not add suffixes of counterexamples to the set of suffixes but rather adds prefixes of counterexamples to the set of prefixes. A detailed account of the correctness of these approaches is omitted here but can be found in [7]. The strategy known best in this class is Angluin's L^* algorithm presented in [7]. It adds all prefixes of a counterexample to the set of prefixes (more specifically: to the set of access sequences). It thus can be seen as the prefix-based version of the **AllGlobally** strategy: all prefixes of a counterexample are used in combination with a global set of suffixes.

A prefix-based strategy resembling **OneLocally** is presented by Kearns and Vazirani in [60]: They determine the exact prefix of a counterexample leading to inconsistency, use only this prefix, and resolve the inconsistency by a suffix that is used only locally (to split one leave in their version of a discrimination tree).

The other two combinations, i.e, adding one prefix of a counterexample while using a global set of suffixes, and using all prefixes but resolving inconsistencies locally remain to be investigated.

Combining prefix-based strategies for handling counterexamples with the observation pack algorithm would require the extension to components with multiple access sequences and the incorporation of methods for resolving inconsistencies into the learning algorithm.

As discussed in the previous section, the different strategies for handling counterexamples influence how new components are created in the observation pack (line 11 of Algorithm 1). In case of using **AllGlobally** or **OneGlobally**, the sets of suffixes of new components are initialized to equal the (global) set of suffixes of the observation pack. In this case all components use a uniform set of suffixes. In case of **OneLocally** the set of suffixes for a new component contains the suffixes on the path in the discrimination tree, leading from the root to the leaf corresponding to the component.

An estimation of the resulting costs in terms of queries for the different strategies for handling counterexamples is given in Section 2.2.3. The results of a (small) experimental evaluation, including the algorithms from [7] and [60] is presented in Section 2.2.4.

2.2.3 Correctness and complexity

The complete observation pack learning algorithm is shown in Algorithm 4. The set of components is initialized with one component for the empty word, containing all inputs as suffixes. The discrimination tree is initialized with one leaf for the empty word. The algorithm then loops closing the observation pack (Algorithm 3) and performing equivalence queries for intermediate hypothesis models (lines 5-6). Counterexamples are exploited (lines 10-19) as discussed in Section 2.2.2. Once an equivalence query signals success, the final hypothesis is returned (lines 7-9).

The correctness of the algorithm follows from the following properties.

1. Algorithm 3 will always return a well-defined hypothesis automaton.

2. Every counterexample can be decomposed as shown in line 10 of Algorithm 4 and in Theorem 2.

As discussed in Section 2.2.2, all strategies for handling counterexamples will lead to unclosedness in at least one of the components, and thus to at least one new state in the subsequent hypothesis. The algorithm terminates as soon as no further counterexample can be found. Since components are only split if a distinguishing suffix is found, all states in the final model are distinguishable, i.e., the model is the canonical (i.e., smallest) Mealy machine for $[\![SUL]\!]$.

For all strategies of handling counterexamples the number of equivalence queries is bounded by the number of states in the canonical Mealy machine. This, however, is a general upper bound. In practice, the number of actual equivalence queries can differ dramatically (cf. Section 2.2.4).

The different strategies for handling counterexamples result in different worst case measurements for the amount of membership queries needed in the course of learning. Let n be the number of states in the final hypothesis, m be the length of the longest counterexample, and k the size of the set of inputs. The resulting complexity (in terms of membership queries) for the discussed approaches is shown in Table 2.1. As a reference the complexity of the L^* version for Mealy machines is given. The number of membership queries can

Algorithm 4 Observation Pack Algorithm

Input: A set of inputs Σ
Output: A model $\mathcal{H}$ with $[\![\mathcal{H}]\!] = [\![SUL]\!]$

Create component $C_\varepsilon = \langle \Sigma \cup \{\varepsilon\}, \varepsilon, \Sigma, \emptyset \rangle$ ▷ Create Initial component
$C := \{C_\varepsilon\}$ ▷ Init. components
$\mathcal{T} := \langle \{n_\varepsilon\}, n_\varepsilon, \emptyset, \{\varepsilon\}, \{(n_\varepsilon, \varepsilon)\}, \{n_\varepsilon\} \rangle$ ▷ Init. discrimination tree
loop
  $\mathcal{H} := ClosePack(\mathcal{T}, C)$ ▷ Hypothesis construction
  $w_c := EQ(\mathcal{H})$ ▷ Hypothesis verification
  if w_c = 'OK' **then** ▷ Counterexample?
    return $\mathcal{H}$ ▷ Then: done
  end if
  Split w_c into uav s.t. $C_u \in C$ and $MQ(uav) \neq MQ(u'v)$, where $ua \in C_{u'}$
  if AllGlobally **then** ▷ Exploit counterexample ...
    Add all suffixes of w_c to V of all $C \in C$
  end if
  if OneGlobally **then**
    Add v to V of all $C \in C$
  end if
  if OneLocally **then**
    Add v to V of $C_{u'} \in C$
  end if
end loop

be estimated as the sum of membership queries needed to complete all components, and the membership queries needed to analyze counterexamples. Membership queries used for sifting new words into the observation pack need not to be considered as they contribute to completing components.

Table 2.1: Membership query complexity for different variants of the observation pack algorithm

Algorithm	Max. table size[1]	Membership queries
Angluin	$nkm \cdot (n + k)$	$O(n^2k \cdot m + k^2n \cdot m)$
Pack + AllGlobally	$nk \cdot (nm + k)$	$O(n^2k \cdot m + k^2n)$
Pack + OneGlobally	$nk \cdot (n + k)$	$O(n^2k + k^2n + n \cdot log_2(m))$
Pack + OneLocally	$nk \cdot (n + k)$	$O(n^2k + k^2n + n \cdot log_2(m))$

As can be seen in Table 2.1, the approaches for handling counterexamples result in a different influence the length of counterexamples has in the worst-case estimation of the number of membership queries. For all approaches the number of suffixes needed to distinguish all states is bounded by n. Adding k suffixes that are in the set of suffixes initially, the size of the set of suffixes can be estimated by $n + k$ for all approaches except for **All-**

[1] For the observation pack this is the accumulated size of all components, i.e., number of prefixes times number of suffixes for each component.

Globally, where all suffixes of counterexamples are added to the set of suffixes. The set of prefixes, on the other hand, can be estimated by kn, i.e., the number of transitions in the final model for all cases except for the L^* algorithm (Angluin), which adds all prefixes of a counterexamples to the set of access sequences.

In the literature k is often assumed to be a small constant and is omitted from the estimation of the membership complexity. The different influence of m in Angluin and **AllGlobally**, which becomes apparent only when including k, however, shows that for practical scenarios where often k dominates n it is important to include k in this estimation.

Finally, assuming a binary search over the length of the counterexample for the correct decomposition, the costs for analyzing a counterexample can be estimated by $log_2(m)$, resulting in the estimations for **OneGlobally** and **OneLocally**. As for equivalence queries, both approaches share the same worst case complexity. In practice, however, the numbers of actual membership queries differ significantly.

Though presented here for Mealy machines, the observation pack algorithm can be used to infer regular languages, too. The only modification that is necessary concerns the initialization of the set of suffixes, which will be initialized by the singleton set containing ε, distinguishing accepted and not accepted words. The estimation of the membership complexity has to be modified accordingly.

Implementing the observation pack algorithm in practice, several optimizations to the version presented here can be realized when using **OneLocally** or **OneGlobally** for handling counterexamples.

1. Counterexamples often can be exploited more than once. In practice, counterexamples can be reused until they are no longer counterexamples instead of performing an equivalence query every time the observation pack is closed.

2. The proof of progress presented here does not ensure that intermediate hypothesis models agree with all made observations. This can easily be enforced using a suffix-closed set of distinguishing suffixes [7]. In [82] (Paper I) an optimization, namely *semantic suffix-closedness*, is presented that ensures that all hypothesis models agree with all observations by closing the set of distinguishing suffixes under suffixes relevant to this end. This, too, can result in a reduced number of equivalence queries.

An implementation of the observation pack algorithm incorporating these optimizations has been realized in LearnLib [67], a library supporting the implementation and evaluation of automata learning algorithms.

2.2.4 Evaluation

In order to evaluate the influence of different strategies for handling counterexamples on the number of membership queries, a series of experiments on models from the Edinburgh Concurrency Workbench (CWB) [68] and from the CADP tool set [34] has been conducted.

The examples from these tool sets are nondeterministic incomplete finite automata with internal actions. In order to infer (DFA) models of these examples, all actions are made inputs, and the examples are transformed into DFAs by means of subset construction. In order to obtain complete models, a non-accepting sink state, looping itself for all inputs, is

added. All inputs that are undefined at a certain state (missing transitions) are directed to this sink. Using DFAs as black-box systems, the DFA version of the observation pack algorithm is used in the experiments. Counterexamples were found by means of the equivalence test from [43].

The experiments subdivide into two series. In one series the observation pack algorithm is combined with the different strategies for handling counterexamples presented in Section 2.2.2. To provide meaningful results, a second series is included in which the same examples are inferred using well-known active learning algorithms from the literature. These algorithms have not been implemented by the author but are taken from libalf [20] (version 0.3), which provides implementations of these algorithms.

The observation pack algorithm has been implemented in LearnLib, which also provides an interface to the algorithms of libalf. The presented figures are from experiments using the observation pack implementation and the libalf implementations of Angluin's L^* algorithm [7], its variant from [62] (referred to as **Maler**) adding all suffixes of counterexamples to the table, and the discrimination tree algorithm [60] (referred to as **Kearns**). In order to provide a fair comparison, only the number of distinct membership queries is counted for all algorithms. This was achieved by using a cache, storing all membership queries used during one experiment. The experiments were conducted using LearnLib on a 2,4GHz AMD Opteron processor with 16 cores and 256GB memory running Linux.

Table 2.2: Results for L^*-based algorithms

Model			Maler		Angluin		Kearns	
Name	$\|Q\|$	$\|\Sigma\|$	MQ	EQ	MQ	EQ	MQ	EQ
vmnew (CWB)	26	4	2,316	8	1,690	9	290	26
cspprot (CWB)	43	5	5,578	8	2,942	9	545	43
peterson2 (CADP)	50	18	63,023	14	18,744	14	1,291	50
sched4 (CWB)	97	12	95,148	13	21,428	12	2,409	97
sched5 (CWB)	241	15	401,865	15	76,914	13	8,869	241
pots2 (CADP)	664	32	7,925,888	53	2,330,380	77	62,021	664

Table 2.2 displays the results for the *libalf* algorithms, Table 2.3 shows the results for the observation pack algorithm. It can be seen, that the performance of **Maler** corresponds exactly to the performance of **AllGlobally**, which is little surprising since both are implementations of the same algorithm. Both use the same number of equivalence queries all experiments.

While it has been argued in Section 2.2.2 that **Angluin** is the prefix-based counterpart of **AllGlobally**, in the experiments its performance shows a much closer correlation to OneGlobally. This can partly be contributed to the particular implementation of the equivalence test, which yields "optimal" counterexamples with prefixes from a learning algorithm's set of prefixes and shortest (new) suffixes. For such counterexamples adding all new prefixes of a counterexample to the set of prefixes is unlikely to result in the worst-case of actually adding all of its prefixes. On the other hand, in practice one can search for such optimal counterexamples actively (cf. Section 3.2).

When inferring DFA models, the observation pack with **OneGlobally** corresponds to the reduced observation table from [75]. Comparing **Angluin** and **OneGlobally**, the results show savings between 10% and 50% in membership queries but for most of the cases an increase in the number of equivalence queries. Though the L^* algorithm is better known and by far more popular than the reduced observation table, the latter is the more efficient algorithm in practice.

Finally, there is a close correlation between **Kearns** and **OneLocally** which both use local sets of suffixes. These two, however, are not implementations of the same algorithm. The **Kearns** algorithm sifts prefixes of counterexamples into the used discrimination tree in order to determine a sensible decomposition of the counterexample. This is more expensive than a binary search over the length of a counterexample and accounts for the slightly increased number of membership queries in **Kearns**. While for **Kearns** the number of equivalence queries in all experiments is identical to the number of states of the SUL, this is not the case for **OneLocally** implementing the optimizations discussed in the previous section.

Table 2.3: Results for Observation pack-based algorithms

Model			AllGlobally		OneGlobally		OneLocally	
Name	$\|Q\|$	$\|\Sigma\|$	MQ	EQ	MQ	EQ	MQ	EQ
vmnew (CWB)	26	4	2,142	8	886	12	250	21
cspprot (CWB)	43	5	4,466	8	2,130	10	509	36
peterson2 (CADP)	50	18	64,099	14	15,674	18	1,314	49
sched4 (CWB)	97	12	98,587	13	17,361	15	2,402	94
sched5 (CWB)	241	15	411,465	15	68,021	19	8,804	235
pots2 (CADP)	664	32	7,906,166	53	1,463,852	68	61,090	581

While **OneLocally** consumes significantly less membership queries than **OneGlobally** in all experiments, it has often been argued that using this strategy for handling counterexamples is not a wise choice in practice since the number of equivalence queries increases drastically. In the next chapter, on the other hand, an approach for organizing equivalence queries is presented that can be described as performing only one incremental equivalence query per learning experiment while searching for counterexamples.

3 Localized inference

In the previous chapter the observation pack algorithm was presented along with different strategies for handling counterexamples. These strategies were shown to be a first means of reducing the number of membership queries needed to infer models of black-box systems. The reduction, however, comes at a price: an increase in the number of equivalence queries. The central idea was applying suffixes of counterexamples only locally, i.e., adding one suffix to one component. In this chapter it will be discussed, how the idea of locality can be applied to the set of prefixes and to equivalence queries as well. In Section 3.1 an approach for "directed" learning will be presented. Section 3.2 describes how equivalence queries can be replaced by a localized version, which is easier to approximate in practical experiments.

3.1 Local exploration

When inferring models of real systems one is often only interested in a certain aspect of the behavior of the SUL, e.g., one may only be interested in the behavior for a subset of all inputs; maybe the interesting subset even depends of the current state of the SUL. On the other hand, for many systems the assumption of a global set of inputs is not natural. Consider, e.g., an e-commerce web-application that will allow providing shipping details only at a certain stage of a purchase. In such situations a means is needed to reflect this partiality (i.e, state-local sets of inputs) in the learning algorithm.

One common approach is the use of filters that help answering membership queries by providing answers without testing the SUL in case the SUL cannot process a query [65, 9]. Such filters have two major drawbacks.

1. Prefixes that are not executable on the SUL will still be added to the observation pack, leading to further membership queries.

2. Filters do not allow for inferring only certain aspects of a SUL.

Here, a new approach for inferring partial models is presented that does not introduce these problems.

3.1.1 Inference with a learning aspect

The observation pack algorithm can be extended easily to inferring partial models by making line 10 in Algorithm 3 conditional: new prefixes are only sifted into the pack if they are relevant to the aspect to be inferred. Technically, this can be realized by a learning aspect.

Definition 7 (Learning aspect) *An aspect for inferring a Mealy machine* $\mathcal{M} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *is a DFA*[1] $P = \langle Q_P, q_0, \Sigma_P, \delta_P, F_P \rangle$*, where*

[1] For a definition of a DFA cf. [44].

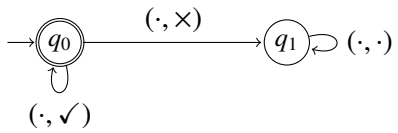

Figure 3.1: Aspect for inferring a partial model of the 3-place buffer from Figure 2.1

- *the initial state q_0 is in the set of final states F_P,*
- *the set of inputs $\Sigma_P = \Sigma \times \Omega$ contains all combinations of inputs and outputs of $\mathcal{M}$, and*
- *every non-final state is a sink: For $q \in Q_P \setminus F_P$ all transitions are reflexive $\delta_P(q, a) = q$ for $a \in \Sigma_P$.*

When describing purposes, symbols $(\cdot, o)$, $(a, \cdot)$, and $(\cdot, \cdot)$ are used as wildcards for all pairs with output o, all pairs with input a, and all combinations of inputs and outputs of $\mathcal{M}$. □

In order to determine if a word $w \in \Sigma^*$ is accepted by an aspect, a membership query is used for every prefix of w. The results are combined into a sequence $(a_1, o_1) \dots (a_n, o_n)$, where $w = a_1 \dots a_n$, in the obvious way.[2] The sequence of input/output pairs can be tested for acceptance by the aspect.

A learning aspect is used to determine whether new prefixes are relevant for the inferred model or not. Only prefixes that lead to input/output sequences accepted by a aspect will be added to the observation pack. When using a learning aspect, the resulting models are partial. Equivalence queries will have to be modified to decide (partial) equivalence of hypothesis models and the system under learning. In fact, this corresponds to using a preorder instead of equivalence, which is common in testing (cf. [22]). The inferred model then corresponds to a specification and the SUL to an implementation.

Figure 3.1 shows a learning aspect for inferring a partial model of the 3-place buffer from Figure 2.1. The aspect will accept all words where none of the prefixes produces an output $\times$ on the buffer. Figure 3.2 shows the corresponding inferred partial model of the buffer: the model contains only transitions that have output $\checkmark$.

The approach of using a learning aspect is a general framework for restricting inference based to some aspect of interest. It thus is different from the application of filters, which primarily aims at reducing the number of membership queries answered by the SUL when inferring complete models. Learning aspects, on the other hand, aim at inferring partial models.

3.1.2 Evaluation

To provide a concrete example, the models used in the experiments presented in Section 2.2.4 all accept prefix-closed languages. Partial models for these prefix-closed languages can be obtained by removing the non-accepting sink state. The resulting (partial) models will only have accepting states.

[2] In practice this usually can be done by a single test on a SUL since a SUL will provide an output after every step in a test.

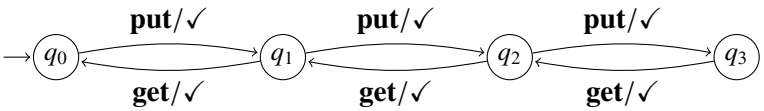

Figure 3.2: Partial model of 3-place buffer

In the series of experiments presented here, these partial models are inferred using a simple test for language containment as learning aspect, resembling the aspect shown in Figure 3.1. This in particular means that the inferred models are of the same size as the target models. Thus, the experiments presented here do not evaluate rigorously how inferred models can be restricted to certain aspects of interest in practice. They are rather to be understood as a proof of concept of inferring partial models.

The experiments were conducted using a modified version (for partial models) of the algorithm from [43] for testing equivalence of automata. Results from this experiments with a learning aspect for exploiting prefix-closedness are presented in Table 3.1 and Table 3.2.[3] The tables compare membership queries and equivalence queries with and without applying a learning aspect. The last two columns of each table show the ratio between the numbers in both cases.

Table 3.1 shows the results of applying the aspect to the observation pack algorithm, using **OneGlobally** as strategy for handling counterexamples. As can be seen in the last two columns, the number of membership queries is reduced significantly by the learning aspect. The number of equivalence queries is not influenced, which is little surprising since, as discussed above, the used aspects do not reduce the number of states in the final models (in this particular case).

Table 3.1: Results for partial exploration (OneGlobally)

Model			complete		partial		$\frac{\text{partial}}{\text{complete}}$	
Name	$\lvert Q\rvert$	$\lvert\Sigma\rvert$	MQ	EQ	MQ	EQ	MQ	EQ
vmnew (CWB)	26	4	886	12	468	12	0.53	1.00
cspprot (CWB)	43	5	2,130	10	721	10	0.34	1.00
peterson2 (CADP)	50	18	15,674	18	1,810	18	0.12	1.00
sched4 (CWB)	97	12	17,361	15	3,498	15	0.20	1.00
sched5 (CWB)	241	15	68,021	19	13,270	19	0.20	1.00
pots2 (CADP)	664	32	1,463,852	68	218,678	68	0.15	1.00

Table 3.2 shows the results of applying the learning aspect to the observation pack algorithm, using **OneLocally** as strategy for handling counterexamples. In this case, neither the number of membership queries nor the number of equivalence queries is influenced by the learning aspect. This is little surprising. Since the aspect was realized by means of member-

[3]The experiments have been conducted on a 2,4GHz AMD Opteron processor with 16 cores and 256GB memory running Linux.

ship queries, the membership query used to determine whether a word should be sifted into the pack corresponds exactly to the membership query spent at the root node of the discrimination tree, deciding whether the word is sorted to the component of the sink state. The corresponding component will have only one suffix during learning. In the case of the **One-Globally** strategy, on the other hand, this component is extended by all newly found suffixes in the course of learning, producing large numbers of in-effective membership queries.

Table 3.2: Results for partial exploration (OneLocally)

Model			complete		partial		$\frac{\text{partial}}{\text{complete}}$	
Name	$\lvert Q\rvert$	$\lvert\Sigma\rvert$	MQ	EQ	MQ	EQ	MQ	EQ
vmnew (CWB)	26	4	250	21	246	21	0.98	1.00
cspprot (CWB)	43	5	509	36	504	36	0.99	1.00
peterson2 (CADP)	50	18	1,314	49	1,296	49	0.99	1.00
sched4 (CWB)	97	12	2,402	94	2,390	94	1.00	1.00
sched5 (CWB)	241	15	8,804	235	8,789	235	1.00	1.00
pots2 (CADP)	664	32	61,090	581	61,058	581	1.00	1.00

In this particular case, the effect of using a learning aspect on the number of membership queries is comparable to the one of using a filter for prefix-closedness [65]. However, there are two important differences. First, in case of using an aspect irrelevant words are not added to the observation pack, which will lead to much less memory consumption and thus allow for inferring bigger models (cf. Section 3.2.3). Second, the inferred models are partial, i.e., have different sets of inputs per state, and thus are more adequate for describing reactive systems in practice.

3.2 Incremental equivalence queries

One problem when applying active automata learning in practice is intrinsic to the MAT learning model: equivalence queries do not exist for black-box systems. They have to be approximated by membership queries, and the possibility of not having tested thoroughly enough will always remain. Due to the close correspondence of equivalence queries and conformance tests, which is discussed in detail in [12], conformance test methods have been suggested for approximating equivalence queries in practice. There are few conformance tests, however, that deal with additional states in the implementation, which in the context of learning is the most interesting phenomenon (e.g., W-method [24] and Wp-method [33]). These methods require the specification of an upper bound to the number of states of the SUL and produce membership queries exponential in the difference between this bound and the number of states of a hypothesis. In order to organize equivalence queries in an improved way, to observations are essential.

First, conformance tests aim at proving conformance between specification and implementation. During learning, however, all but one equivalence query are used to find counterexamples. Thus, in the vast majority of cases, an efficient method for finding counterex-

amples is required rather than a method for rigidly proving absence of counterexamples. One idea for how to search for counterexamples is provided by Theorem 2. Since every counterexample will be decomposed into a prefix from the set of access sequences in the observation pack, an input, and a suffix, it may be beneficial to use candidate counterexamples of this form.

Second, in the MAT model different equivalence queries are independent. Each query takes a model and tests the equivalence of this model and the SUL. Also, the formulation of the classic L^* algorithm does not make it obvious if or how subsequent hypothesis models are related. Subsequent hypothesis models produced by the observation pack algorithm, on the other hand, are strongly related. The set of access sequences computed by the algorithm is a prefix-closed set of words. As an analogy, this set can be thought of an incrementally growing spanning tree covering the state space of a SUL. Intuitively, the set of all prefixes extends this spanning tree by all transitions that turn the spanning tree into an automaton.

Using the spanning tree, i.e., the set of access sequences, subsequent hypothesis models can be related, as is shown schematically in Figure 3.3. The figure will be discussed in detail below. For now, let $\{\varepsilon, a, aa\}$ be the spanning tree that evolves from $\mathcal{H}_1$ to $\mathcal{H}_3$. The hypothesis evolves by adding one of the transitions to the tree that is not yet part of the spanning tree (marked (?) in the figure), and adding some new transitions originating in the corresponding new state.

While in the sequence shown in 3.3 there is only one unconfirmed transition in each hypothesis (marked (?)), and only this transition is added to the spanning tree, in general there will be many such unconfirmed transitions, and in a single round of learning more than one of these may be added to the spanning tree. Yet, especially when using **OneLocally** for handling counterexamples and inferring models of considerable size, changes between two subsequent hypothesis models will be very local.

Taking these observations into account, the obvious conclusions are that (1) equivalence of models can be subdivided into equivalence of single transitions, and that (2) only new or modified transitions have to be tested in the process. Let thus the MAT model be modified by providing a learning algorithm with membership queries and identity queries.

Identity queries test whether two prefixes u, u' are in the same class of $\equiv_{SUL}$. In case $u \not\equiv_{SUL} u'$ they return a suffix v for which $[\![SUL]\!](uv) \neq [\![SUL]\!](u'v)$. Otherwise they return successfully.

During learning, identity queries will be used to test whether a prefix from one component in the observation pack leads to the same state in the SUL as its access sequence (from the same component). Of course, in practice identity queries are no less unrealistic than equivalence queries, but as will be argued in Section 3.2.2 and Section 3.2.3 they provide a framework for organizing equivalence queries incrementally and can be approximated easily.

3.2.1 The evolving hypothesis algorithm

The evolving hypothesis algorithm, which replaces the equivalence oracle in learning setups, works in two phases. In the first phase the hypothesis model maintained by the algorithm is updated using the most recent hypothesis produced by the learning algorithm

Algorithm 5 Test evolving hypothesis($\mathcal{H}$, $\mathcal{H}_E$, W)

Input: Current hyp. $\mathcal{H}$, evolving hyp. $\mathcal{H}_E$, and a list of words W
Output: A counterexample or 'OK'

```
1:  for q ∈ Q_H and a ∈ Σ do                          ▷ update modified transitions
2:      u := ⌊q⌋_H (i.e., the known access sequence of q)   ▷ trace for state
3:      ua := u · a                                   ▷ trace for transition
4:      if ua ≠ ⌊ua⌋_H and ⌊ua⌋_H ≠ ⌊ua⌋_H_E then     ▷ No access seq. and modified?
5:          W := W ∪ {ua}                             ▷ Then: mark for checking
6:      end if
7:  end for
8:  H_E := H                                          ▷ 'evolve' hypothesis
9:  while W ≠ ∅ do                                    ▷ check modified transitions
10:     u := poll(W)                                  ▷ Remove candidate from W
11:     u' := ⌊u⌋_H_E                                 ▷ Get access seq. for destination
12:     w_c := IQ(u, u')                              ▷ Perform IQ
13:     if w_c ≠ 'OK' then                            ▷ Found a counterexample?
14:         W := W ∪ {u}                              ▷ Then: re-add prefix to work list
15:         return w_c                                ▷ and return counterexample
16:     end if
17: end while
18: return 'OK'                                       ▷ Done
```

and the set of access sequence from the observation pack. In the second phase, the evolved hypothesis is tested for equivalence to the SUL using identity queries.

The algorithm is shown in Algorithm 5. It is assumed that for a hypothesis $\mathcal{H}$ it is known which access sequence belongs to which state. The algorithm uses an evolving hypothesis $\mathcal{H}_E$ and a queue W of unconfirmed transitions, i.e., prefixes from the observation pack that are not in the set of access sequences. Upon first invocation (not shown in the pseudo-code), the first half of the algorithm is skipped (lines 1-7). Instead, all prefixes that are not access sequences are simply added to W. In subsequent invocations all transitions that have been modified, i.e., that lead to different states in $\mathcal{H}$ and $\mathcal{H}_E$ are added to the queue of unconfirmed transitions unless they belong to the spanning tree (lines 1-7). As indicated in lines 6 and 14 of the algorithm, the queue of unconfirmed transitions contains every transition of $\mathcal{H}$ at most once. After updating the set of transitions to be verified, $\mathcal{H}_E$ is evolved, i.e., replaced by $\mathcal{H}$ (line 8).

In the second phase transitions in W are checked by means of identity queries (lines 9-17). For every transition it is checked whether the corresponding prefix leads to the same state in SUL than its access sequence in the hypothesis. In case the identity query (line 14) returns a counterexample, the prefix is re-added to the set of unconfirmed transitions and the counterexample is returned. Re-adding the prefix is necessary since it is not guaranteed how the counterexample is used by the learning algorithm. This will be discussed below. In case no counterexample is found for any of the unconfirmed transitions, the algorithm terminates successfully (line 18). The work list W is not emptied between subsequent invocations of the algorithm.

Figure 3.3 shows schematically an evolving hypothesis. Unconfirmed transitions in the subsequent hypothesis models are marked by (?), while confirmed ones are marked by (✓).

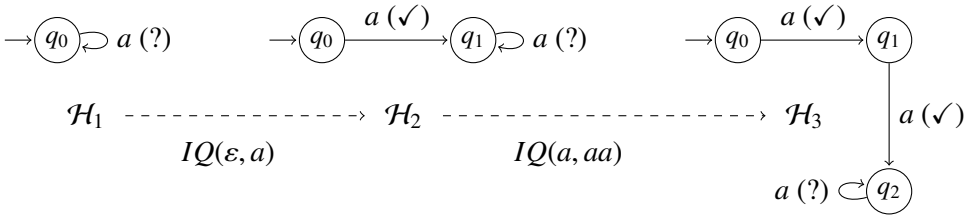

Figure 3.3: Sequence of evolving hypothesis models

Identity queries (and intermediate rounds of learning) lead to the refinements shown in the figure. In the first step, the unconfirmed transition for a is refined into a transition to a new state, a becoming the access sequence of this new state. In the second step the same is done for the prefix aa.

3.2.2 Correctness and complexity

The correctness of the presented algorithm is straightforward. Transitions of a hypothesis are added to the work list of the algorithm whenever their destination changes. When a transition is removed from the work list an identity query is performed for the corresponding prefix ua and its access sequence u' (unless ua is an access sequence already).

If the identity query for ua and u' returns a suffix v, either uav or $u'v$ is a counterexample since

$$[\![\mathcal{H}]\!](uav) = [\![\mathcal{H}]\!](u'v) \qquad \text{while} \qquad [\![SUL]\!](uav) \neq [\![SUL]\!](u'v).$$

The counterexample is passed to the learning algorithm and ua is added to the work list again.[4] An identity query may lead to removing a transition from the work list permanently or to a counterexample. Counterexamples can be found at most $n - 1$ times. On the other hand, $n - 1$ transitions are access sequences and will not be tested. This yields.

Theorem 3 *Regular languages and regular sets of traces can be inferred with polynomially many membership queries and $O(kn)$ identity queries.* □

Algorithm 5 will not only work with the observation pack algorithm but with any learning algorithm that uses a growing set of unique access sequences to states of the hypothesis.

Of course, for black-box systems identity queries can not be computed. They have to be approximated. The concrete realization of such an approximation follows straightforwardly from the idea of the queries itself. One only needs to provide candidate suffixes. As for equivalence queries, the possibility of not having tested enough will always remain in the black-box scenario. Though, identity queries provide a valuable framework for organizing

[4] A learning algorithm does not necessarily use this counterexample to split the component of ua and u'. Using a strategy like *OneLocally* for analyzing counterexamples may lead to a different component being split if the counterexample provides a reason for splitting that component, too. Even if the component of ua and u' is split, it is not guaranteed that ua becomes the access sequence of the newly created component. This could be realized by tighter integration of the evolving hypothesis algorithm and the learning algorithm. On the other hand, such an integration would make the interface between learning algorithm and equivalence test less general. Realizing the algorithm in LearnLib, generality outweighed performance in this particular decision. Thus, it is necessary to re-add ua to the work list, such that u' and ua can be tested again in a subsequent identity query.

equivalence queries. They provide a pattern for the search for counterexamples in practical applications, while still realizing an (incremental) conformance test.

An approximative version of the evolving hypothesis algorithm has been used in the ZULU competition [26]. Combined with the observation pack algorithm and the **OneLocally** strategy for handling counterexamples it outperformed all other solutions for approximating equivalence queries by means of membership queries. Detailed results are discussed in [50] (Paper II).

3.2.3 Evaluation

The evolving hypothesis algorithm has been evaluated on the set of examples from Section 2.2.4. In order to investigate the relation between equivalence queries and identity queries, two series of experiments have been performed using the observation pack algorithm. In one series **OneGlobally** is used to handle counterexamples, while in the other series counterexamples are analyzed by the **OneLocally** strategy. Algorithm 5 was used to realize equivalence queries. Identity queries have been realized by a synchronized breath-first search (starting in different states) on the target systems. In each experiment the overall number of these equivalence queries and the total number of identity queries were observed. Results of all experiments are shown in Table 3.3.[5]

Table 3.3: Equivalence vs. Identity Queries

Model			OneGlobally		OneLocally		$\frac{\text{OneLocally}}{\text{OneGlobally}}$	
Name	$\lvert Q\rvert$	Trans.	EQ	IQ	EQ	IQ	EQ	IQ
vmnew (CWB)	26	104	11	89	20	98	1.82	1.10
cspprot (CWB)	43	215	10	182	30	202	3.00	1.11
peterson2 (CADP)	50	900	18	868	49	899	2.72	1.04
sched4 (CWB)	97	1,164	15	1,082	92	1,159	6.13	1.07
sched5 (CWB)	241	3,615	18	3,392	228	3,602	12.67	1.06
pots2 (CADP)	664	21,248	60	20,644	568	21,152	9.47	1.02

While the increase in equivalence queries ranges from about 80% to almost 1,200% between the two series, the amount of identity queries increases only moderately by 2% to 11%. Identity queries almost being constant, the average number of identity queries per equivalence query mainly depends on the number of equivalence queries. Put differently, the average costs of equivalence queries are not uniform for the same experiment in both series of experiments.

The overall number of identity queries during a learning experiment on the other hand is almost invariant under the application of different strategies for handling counterexamples. This is not surprising. It is a direct consequence of Theorem 3, and suggests that the number of identity queries is a more appropriate measure than the number of equivalence queries

[5] The experiments have been conducted on a 2,4GHz AMD Opteron processor with 16 cores and 256GB memory running Linux.

when comparing different strategies for handling counterexamples. The less equivalence queries are used during learning the more complex are the single equivalence queries.

Finally, to explain the increase in identity queries, please observe that for every row of Table 3.3 for both series of experiments the following equation holds

$$\text{IQ} \;=\; \text{Trans.} - (|Q| - \text{EQ}).$$

The number of identity queries corresponds exactly to the number of transitions minus the number of states that has been found without equivalence query. Of the $|Q| - 1$ access sequences $|Q| - 1 - (\text{EQ} - 1)$ many are found without consuming identity queries: The potential increase in the number of identity queries between **OneGlobally** and **OneLocally** is limited by the number of states in the final model.

Finally, the integration of local exploration and local equivalence tests (i.e., the evolving hypothesis algorithm) has been tested in an additional series of experiments on bigger examples from the CADP tool set. Automata models have been extracted from the examples as described in Section 2.2.4. In all experiments the observation pack algorithm is combined with the **OneLocally** strategy for handling counterexamples, a learning aspect resembling the one in Figure 3.1 exploiting partiality of the models, and the evolving hypothesis algorithm. The results of the experiments are presented in Table 3.4.

Table 3.4: Combining local exploration and local equivalence

Model			Results			
Name	$\|Q\|$	$\|\Sigma\|$	Trans.	MQ	EQ	IQ
co4-3-1	9,979	85	21,237	2,171,464	6,433	17,691
turntable_par	22,846	25	82,983	2,759,885	18,691	78,828
SCSI_A	56,168	28	154,748	6,899,646	35,030	133,610
co4-4-1	166,883	121	483,735	47,090,040	106,358	423,210

To the author's knowledge, models of this size have not been inferred using active automata learning methods before. The biggest inferred (non partial) model is reported in [73], namely a router with 11 inputs and about 22,000 states.

To verify that the increased performance is not only due to newer hardware and more available memory, execution times and memory consumption were recorded in the experiments. In the experiment "turntable_par" 0.36GB of memory was used and learning required 197 seconds, while identity queries consumed 16 seconds. In the case of "co4-4-1" learning finished in 3 hours and 50 minutes using 4.05GB of memory. Identity queries consumed 58 seconds.

4 Active learning of interface programs

The previous two chapters presented the central ideas for learning models of black-box systems by means of membership queries and equivalence queries. This chapter will focus on extending active learning to interface programs.

The main problem when inferring interface programs for real black-box systems by means of active automata learning is that DFAs or Mealy machines are not designed to describe such systems: Usually, real black-box systems do not only consist of a control structure, which can be described faithfully by an automaton, but also use data: Inputs of such systems comprise data values. Making matters worse, these data values typically influence the control behavior. As a simple example, consider an identifier for a session in a web-application. The behavior exposed by the application will strongly depend on the concrete session identifier that one includes in the communication with this application. Other examples of data parameters that have an impact on the behavior of a systems are authentication credentials and sequence numbers. Providing valid credentials usually leads to different behavior than providing invalid ones. The same holds for sequence numbers: one has to follow a strict pattern (i.e., increasing at the right moment of time) in order to achieve progress in some protocol. Thus, when extending regular inference to systems with data, two problems arise.

1. Data values usually are from unbounded domains (e.g., natural numbers or strings), leading to infinite sets of inputs.

2. Mealy machines and DFAs are adequate models of control behavior, but these automata do not model explicitly data parameters or their influence on the behavior. In order to model such an influence, storing data values and comparing stored data values has to be made explicit.

Both problems, infinity and expressivity, have to be addressed twice: once at the level of an adequate modeling formalism, and once at the level of a learning algorithm. In particular, active learning relies on the following ideas, which have to be adapted.

1. A Nerode-like relation allowing the suffix-based identification of elements in inferred models, i.e., states and transitions (Section 2.1).

2. A Myhill/Nerode-like theorem establishing a method for constructing automata from regular semantic functionals (Section 2.1).

3. A data structure from which at certain points in time hypothesis automata can be constructed (Section 2.2.1). In particular this data structure has to realize the Nerode-relation using finite sets of prefixes and suffixes.

4. A method for exploiting counterexamples, ensuring progress (Section 2.2.2).

In the next section, regular inference is extended to systems with infinite sets of inputs. Section 4.2 presents a modeling formalism and a corresponding Nerode-relation for a class of systems that store data values. This class of systems can be inferred using finite sets of prefixes and suffixes, as will be discussed in Section 4.3. In both cases an extended method for analyzing counterexamples will be necessary. Finally, in Section 4.4 the ideas from the previous sections will be extended to systems that can produce parameterized output using stored data values together with data from a small evaluation of the approach.

4.1 Automated alphabet abstraction refinement

As discussed above, real systems usually use inputs with data parameters over infinite domains. The resulting set of inputs is infinite. As an example consider a threshold for some sensor in a reactive system. The behavior of the system will only depend on a data value exceeding the threshold value or not. Figure 4.1 details this example. Inputs **set**(x) with $x \in \mathbb{Z}$ set the value of the sensor. Input **test** reads the value of the sensor afterwards. For values below or equal 10, reading will produce a positive output (i.e., ✓). Sensor values above the threshold will lead to output × on reading.

Inferring a model of the sensor would require infinitely many tests when applying active learning directly. Luckily, in many cases (especially if data values from inputs cannot be remembered by a SUL) the data values occurring in inputs can be partitioned into finitely many classes: A model of the behavior can abstract from the concrete data values. The obvious abstraction indicated in the figure is using two abstract **set** operations.

Inferring abstract models directly, however, is difficult for two reasons. First, active learning relies on tests being executed on a SUL, and these tests have to be sequences of concrete inputs. Thus, one needs a system of representative inputs for an abstraction. Second, defining (correct) abstractions prior to learning to some extent contradicts the idea of inference and in most situations is not feasible. The abstract classes of inputs depend on the behavior of a system, which is unknown prior to learning.

In [51] (Paper III) a method avoiding both these problems is presented. By inferring a system of representative inputs during learning and integrating automated alphabet abstraction refinement (AAAR) into the learning algorithm, the algorithm will work at the level of a concrete and finite representation of a SUL. Abstractions can be refined in the course of learning, modeled as monotonic growth of the set of inputs used in the the learning algorithm. This section gives a brief introduction to these ideas and relates them to the four building blocks of active learning enumerated in the introduction to this chapter.

Let Σ_C be a countable set of inputs, and $\mathcal{M}_C = \langle Q, q_0, \Sigma_C, \Omega, \delta, \lambda \rangle$ be a *countable Mealy machine* (CMM), i.e., a Mealy machine with a countable set of inputs. Figure 4.1 shows a countable Mealy machine for the discussed example. Luckily, the inputs from Σ_C of a countable Mealy machine can be partitioned into finitely many classes since a CMM has only finitely many states. This leads to the following definition.

Definition 8 (Equivalent input symbols) *For a regular set of traces T, two input symbols $a, a' \in \Sigma_C$ are equivalent, denoted by $a \simeq_T a'$, if for all $u, v \in \Sigma_C^*$ it holds that $T(uav) = T(ua'v)$.* □

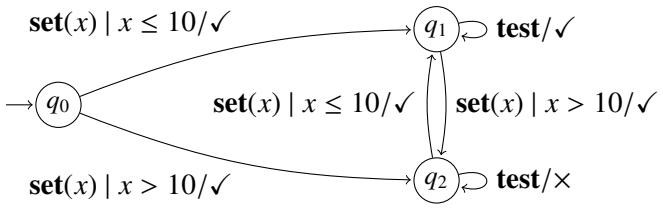

Figure 4.1: Partial countable Mealy machine model of sensor with threshold with infinite set of inputs **test** and **set**(x) for $x \in \mathbb{Z}$.

While the equivalence relation on words $\equiv_T$ is a right-congruence, the equivalence relation on inputs is a middle-congruence. Accordingly, input actions are not distinguished (wrt. $\simeq_T$) by suffixes but by pairs of prefixes and suffixes.

Definition 9 (Witness) *A triple $\langle u, v, o\rangle \in \Sigma^* \times \Sigma^* \times \Omega$ is referred to as a* witness. *A witness proves inputs $a, a' \in \Sigma_C$ inequivalent if $T(uav) = o$ but $T(ua'v) \neq o$ (or vice versa).* □

In the example from Figure 4.1, the witness $\langle \varepsilon, \textbf{test}, \times\rangle$ partitions all inputs **set**(x) with $x \in \mathbb{Z}$ into two classes, one class of inputs below or equal to the threshold, and one class of inputs above the threshold. Including an output in a witness guarantees that every witness partitions the set of inputs into at most two classes.

While in a Mealy machine infinitely many words may lead to the same state, active learning works on a finite prefix-closed set of words, reaching every state (i.e., class of $\equiv_T$) once. A similar approach can be used to infer countable Mealy machine models, using one concrete input for every class of $\simeq_T$, potentially containing infinitely many inputs.

Let W be a finite set of witnesses, and let $a \simeq_W a'$ denote the fact that no witness in W proves the inequivalence of a and a'. Accordingly, for $a \not\simeq_W a'$ there is a witness in W proving the inequivalence of a and a'. A set $\Sigma_W \subset \Sigma_C$ of *representative inputs* is a finite set, where $a \not\simeq_W a'$ for $a, a' \in \Sigma_W$, and such that for every $a'' \in \Sigma_C$ there is a representative input a in Σ_W such that $a \simeq_W a''$. Let then $\rho_W : \Sigma_C \to \Sigma_W$ map every input in Σ_C to its unique representative (i.e., equivalent wrt. $\simeq_W$) input from Σ_W.[1] The mapping ρ_W can be extended to a mapping $\Sigma_C^* \to \Sigma_W^*$ in the obvious way.

Certainly, an active learning algorithm can be used to construct a Mealy machine hypothesis for a finite subset Σ_W of Σ_C, i.e., at the level of the concrete representative inputs. The remaining problem is an equivalence query, which may return counterexamples from Σ_C^*. Counterexamples obtained during learning can either contradict the intermediate abstraction or the intermediate hypothesis over Σ_W. Let $w \in \Sigma_C^*$ be a counterexample, i.e., such that $[\![\mathcal{H}]\!](\rho_W(w)) \neq [\![S\,UL]\!](w)$. Then it can either be the case that

$$[\![S\,UL]\!](\rho_W(w)) \neq [\![S\,UL]\!](w) \qquad \text{or} \qquad [\![S\,UL]\!](\rho_W(w)) \neq [\![\mathcal{H}]\!](\rho_W(w)).$$

In the second case it contradicts the hypothesis over Σ_W. This case is not different to an ordinary counterexample and can be handled as described in Chapter 2. In the first case,

[1] Technically, such a mapping and set of representative inputs can be realized by means of a binary decision tree, in which the inner nodes are labeled with witnesses, and leaves are labeled with representative inputs. The mapping ρ_W can then be computed by sinking inputs into the tree.

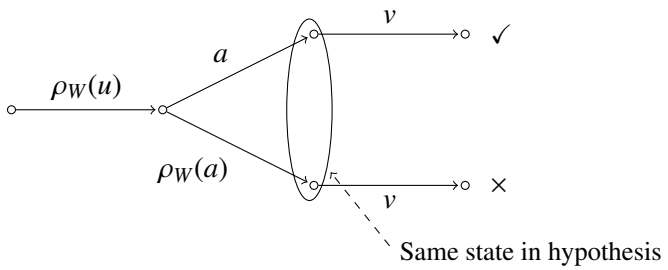

Figure 4.2: Exploiting counterexamples for alphabet abstraction refinement

on the other hand, the counterexample contradicts the abstraction induced by ρ_W. The abstraction has to be refined by means of a witness

Let w be a counterexample of the first category. Then there has to be a smallest index i such that $w = uv$, where u is of length i, and $[\![SUL]\!](\rho_W(u)v) \neq [\![SUL]\!](uv)$. This directly yields the following theorem.

Theorem 4 (Counterexample Decomposition (AAAR)) *For a set of a representative inputs Σ_W, a corresponding mapping ρ_W, and a counterexample $w \in \Sigma_C^+$ with $[\![SUL]\!](w) \neq [\![SUL]\!](\rho_W(w))$) there exists a decomposition $w = uav$ into a prefix $u \in \Sigma_C^*$, an input $a \in \Sigma_C$, and a suffix $v \in \Sigma_C^*$ such that $[\![SUL]\!](\rho_W(u)\, av) \neq [\![SUL]\!](\rho_W(ua)\, v)$.* □

Figure 4.2 shows the situation stated in the theorem schematically. At some point of the counterexample (after $\rho_W(u)$), the input a and its representative input $\rho_W(a)$ lead to different states in the SUL, which is proven by suffix v of the counterexample.

The triple $\langle \rho_W(u), v, o \rangle$ with $o = [\![SUL]\!](\rho_W(u)\, av)$ is a witness for the inequivalence of a and $\rho_W(a)$. Figure 4.3 shows this schematically. The class $\rho_W^{-1}(a)$ is refined by the witness from the counterexample into two classes of a new representation induced by the set W', containing the additional witness. The input a from the counterexample becomes the representative input for the new class $\rho_{W'}^{-1}(a)$.

Starting with an arbitrary input from Σ_C as initial representative input, the approach sketched above can be iterated for a monotonically growing set Σ_W of representative inputs. At every stage of the process, a hypothesis is constructed for Σ_W. Every counterexample either leads to a refined hypothesis for Σ_W, or to an extension of Σ_W.

In [51] (Paper III) it is shown that this approach terminates with an optimal *determinism preserving* abstraction, i.e., one with a minimal set Σ_W for which $[\![SUL]\!](w) = [\![SUL]\!](\rho_W(w))$ for all $w \in \Sigma_C^*$. This overcomes the problem of finding correct abstractions prior to learning, while still polynomially many membership queries and equivalence queries are sufficient to infer models with automated alphabet abstraction refinement.

The approach works for regular sets of traces, where the index of $\equiv_T$ is finite. Regularity makes it easy to provide the following prerequisites to active learning from the list in the introduction to this chapter: (1) The equivalence relation on the set of inputs and witnesses realizing these are used to identify representative transitions in inferred models. (2) Automata construction at the level of representative inputs does not differ from the case of Mealy machines. (3) The finite indices of both equivalence relations guarantee that the learning algorithm can work on finite sets of prefixes and suffixes.

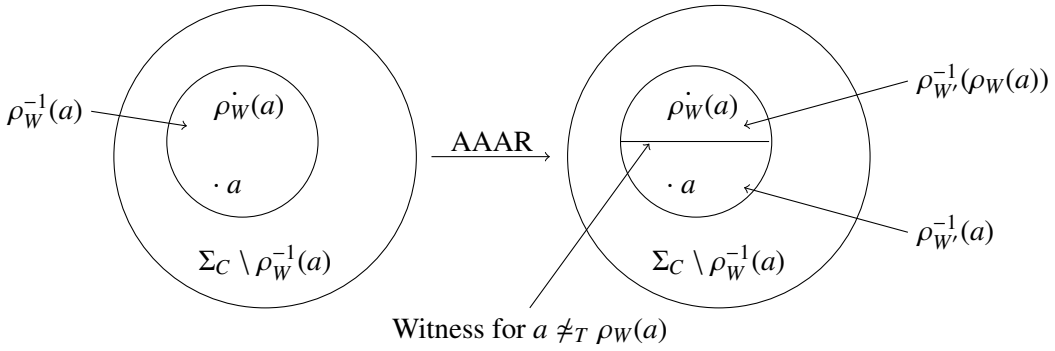

Figure 4.3: Automatic alphabet abstraction refinement for inputs $a, \rho_W(a) \in \Sigma_C$ and $a \not\equiv_T \rho_W(a)$

Finally, the process of analyzing counterexamples has to be extended using a two-step approach for analyzing counterexamples, distinguishing between counterexamples that refine the chosen representation and regular ones that lead to new states. While for the former case a new pattern of analysis is required, the latter case is covered by the pattern presented in Section 2.2.2.

4.2 Modeling systems capable of storing data values

As discussed above, in many real systems data parameters of inputs influence the behavior of the system. This is already true for the class of systems discussed in the previous section. These systems could be characterized using the (classic) Nerode-relation directly. As soon as systems can store data values and compare these stored data values, this in general is not possible. As an example consider a system that allows users to register an account with a set of credentials and afterwards login to this account with the credentials provided during registration. The behavior of such a system will depend on the credentials provided during registration, which can be from some infinite domain. Using the classic Nerode-relation, this directly yields infinitely many classes, one for every concrete set of credentials.

In order to represent such systems as finite models, storing and comparing data values has to made explicit in the automata representation. In this section the results from [23] (Paper IV) will be discussed, presenting an automaton model for a special class of such systems, comparing data values for equality.

Presentation in this section served as a basis for in [46] (Paper VI) and is thus very similar to the paper.

4.2.1 Register automata

In [23] (Paper IV) we have presented register automata as a modeling formalism for interface programs. Let $\mathcal{D}$ be an unbounded domain of data values which can be compared for equality, and Σ be a set of *parameterized inputs*, each with a fixed arity (i.e., number of arguments it takes from $\mathcal{D}$). A *data input* is a pair $(a, \bar{d})$, where $a \in \Sigma$ with arity k, and $\bar{d} = d_1, \ldots, d_k$ is a sequence of data values from $\mathcal{D}$. Sequences of data inputs are *data*

words. For a data word w, let $Acts(w)$ be the sequence of parameterized inputs in w and $Vals(w)$ be the sequence of data values in w (from left to right). Let then $ValSet(w)$ denote the set of distinct data values in $Vals(w)$. Data words are concatenated just like plain words.

Let $\mathcal{W}_{\mathcal{D}}$ be the set of all data words for a fixed set of parameterized inputs and a data domain $\mathcal{D}$, and $\pi : \mathcal{D} \to \mathcal{D}$ be a permutation on $\mathcal{D}$. A permutation π is applied to a data word w pointwisely on $Vals(w)$. On a set of data words, π is applied pointwisely on every data word in the set. Finally, a *data language* is a set of data words $\mathcal{L}_{\mathcal{D}} \subseteq \mathcal{W}_{\mathcal{D}}$ that is closed under permutations on $\mathcal{D}$, i.e., $\mathcal{L}_{\mathcal{D}} = \pi(\mathcal{L}_{\mathcal{D}})$ for any permutation π on $\mathcal{D}$.

As an example for a data language consider the following language over the domain of natural numbers and the set {**reg**, **login**} of parameterized inputs, each of arity two. The language $\mathcal{L}_{login}$ consists of all sequences of **reg** and **login**, where the data values of **login** match the ones of **reg**, i.e,

$$\mathcal{L}_{login} \;=\; \{(\mathbf{reg}, \langle d_1, d_2\rangle)(\mathbf{login}, \langle d_3, d_4\rangle) \mid d_1 = d_3 \;\wedge\; d_2 = d_4\}.$$

Intuitively, one has to login with the credentials one has registered with, and $\mathcal{L}_{login}$ describes valid sequences of register and login. While $(\mathbf{reg}, \langle 1, 2\rangle)(\mathbf{login}, \langle 1, 2\rangle)$ and $(\mathbf{reg}, \langle 3, 3\rangle)(\mathbf{login}, \langle 3, 3\rangle$ are in $\mathcal{L}_{login}$), the data word $(\mathbf{reg}, \langle 1, 2\rangle)(\mathbf{login}, \langle 3, 4\rangle)$ is not.

Let now a *symbolic input* be a pair $(a, \bar{p})$, of a parameterized input a of arity k and a sequence of symbolic parameters $\bar{p} = \langle p_1, \ldots, p_k\rangle$ Let further $X = \langle x_1, \ldots, x_m\rangle$ be a finite set of *registers*. A *guard* is a propositional formula of equalities and negated equalities over symbolic parameters and registers of the form

$$G ::= G \wedge G \mid G \vee G \mid x_i = p_j \mid x_i \neq p_j \mid true,$$

where *true* denotes the atomic predicate that is always satisfied. An *assignment* is a partial mapping $\sigma :\; X \to X \cup P$ for a set P of formal parameters.

Definition 10 (Register Automaton) *A* Register Automaton *(RA) is a tuple* $\mathcal{A} = (\Sigma, L, l_0, X, \Gamma, \lambda)$*, where*

- Σ *is a finite set of* parameterized inputs.
- L *is a finite set of* locations.
- $l_0 \in L$ *is the* initial location.
- X *is a finite set of* registers.
- Γ *is a finite set of* transitions*, each of which is of form* $\langle l, (a, \bar{p}), g, \sigma, l'\rangle$*, where* l *is the* source location*,* l' *is the* target location*,* $(a, \bar{p})$ *is a symbolic input,* g *is a guard, and* σ *is an assignment.*
- $\lambda : L \mapsto \{+, -\}$ *maps each location to either + (accept) or − (reject).* □

Let us define the semantics of an RA $\mathcal{A} = (\Sigma, L, l_0, X, \Gamma, \lambda)$. A *valuation*, denoted by ν, is a (partial) mapping from X to $\mathcal{D}$. A *state* of $\mathcal{A}$ is a pair $\langle l, \nu\rangle$ where $l \in L$ and ν is a valuation. The *initial state* is the pair of initial location and empty valuation $\langle l_0, \emptyset\rangle$.

A *step* of $\mathcal{A}$, denoted by $\langle l, \nu\rangle \xrightarrow{(a,\bar{d})} \langle l', \nu'\rangle$, transfers $\mathcal{A}$ from $\langle l, \nu\rangle$ to $\langle l', \nu'\rangle$ on input $(a, \bar{d})$ if there is a transition $\langle l, (a, \bar{p}), g, \sigma, l'\rangle \in \Gamma$ such that

1. g is modeled by $\bar{d}$ and ν, i.e., if it becomes true when replacing all p_i by d_i and all x_i by $\nu(x_i)$, and

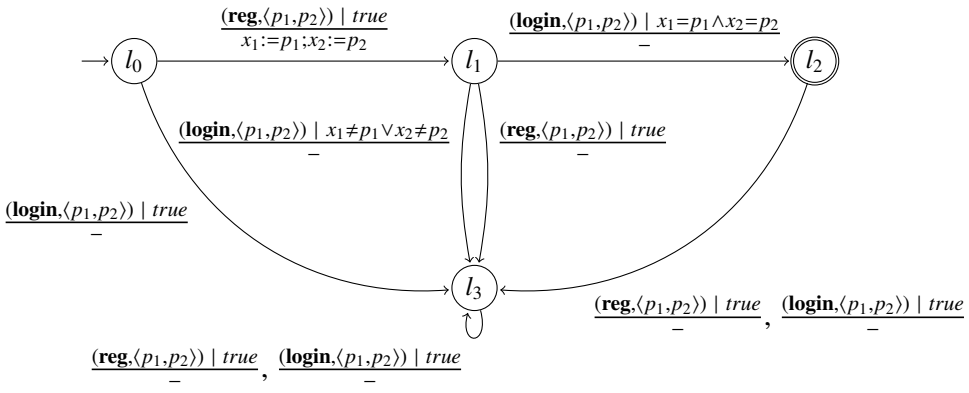

Figure 4.4: Register automaton for $\mathcal{L}_{login}$. Accepting location marked by double circle.

2. ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ wherever $\sigma(x_i) = x_j$, and $\nu'(x_i) = d_j$ wherever $\sigma(x_i) = p_j$.

A *run* of $\mathcal{A}$ over a data word $(a_1, \bar{d}_1) \dots (a_k, \bar{d}_k)$ is a sequence of steps

$$\langle l_0, \emptyset \rangle \xrightarrow{(a_1,\bar{d}_1)} \langle l_1, \nu_1 \rangle \quad \dots \quad \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{(a_k,\bar{d}_k)} \langle l_k, \nu_k \rangle.$$

A run is *accepting* if $\lambda(l_k) = +$, otherwise it is *rejecting*. The data language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of data words that it accepts. A register automaton $\mathcal{A}$ is *determinate* if no data word has accepting and non-accepting runs in $\mathcal{A}$. A data word w is *accepted* by a determinate register automaton (DRA) $\mathcal{A}$ if all runs of w in $\mathcal{A}$ are accepting.[2]

Two registers $x_i, x_j \in X$ of $\mathcal{A}$ are *independent* if the behavior of $\mathcal{A}$ does not depend on the equality (or inequality) of x_i and x_j. Technically, this means that

- no guard of any transition may require the equality of x_i and x_j, and
- no combination of guard and assignment may imply the equality of x_i and x_j.

A register automaton is *right-invariant* if all registers are pairwisely independent.

Right-invariance is important for defining canonical DRAs by means of an extended Nerode-relation for data languages. This allows for using one location for every class of the (to be defined) Nerode-relation since in a right-invariant DRA in no state the future behavior depends on the equality of registers.

Figure 4.4 shows a DRA for $\mathcal{L}_{login}$. From the initial location it is possible to register with a username and password. Then, it is possible to login with the these credentials. All other inputs will lead to the non-accepting sink location l_3.

4.2.2 Suffix-based model construction

In the introduction to this chapter it has been argued that when extending expressivity of the modeling formalism, four problems arise. The first two of these, (1) a suffix-based

[2] Determinacy is a way of encoding disjunctions in guards implicitly (instead of working with explicit disjunctions and determinism). It allows to relate data languages and register automata more easily than explicit disjunctions when constructing canonical automata for data languages.

identification of all parts of a model (locations, transitions, and registers in this case) and (2) a method of generating automata from sets of traces, will be discussed in this section. The remaining two problems (using finite sets of prefixes and suffixes in a learning algorithm and handling counterexamples) will be addressed in the following two sections.

First, in order to identify locations in a register automaton, a corresponding Nerode-relation with finite index is needed. Such a relation alone, however, is not sufficient, when constructing canonical register automata models for sets of data words. One will also need a means of deriving registers, guards, and assignments from data words. As will be shown, suffixes can be used to determine registers and assignments in a canonical automaton. Guards of transitions can be derived from data words when combining the ideas from Section 4.1, i.e., using a system of representative inputs with an order on the inputs.

Identifying locations. Let us begin with the adaption of the Nerode-relation to data languages. Let the *residual* of some word $u \in \mathcal{W}_\mathcal{D}$ wrt. a data language $\mathcal{L}_\mathcal{D}$, denoted by $u^{-1}\mathcal{L}_\mathcal{D}$, be the set of words v with $uv \in \mathcal{L}_\mathcal{D}$. As discussed above, the set of data inputs over $\mathcal{D}$ and some set of parameterized inputs Σ is infinite. Unlike in the case of countable Mealy machines (Section 4.1), however, also the (classic) Nerode-relation on words over this set of inputs will have infinitely many classes in the case of data languages. In the example of $\mathcal{L}_{login}$, the data words $(\mathbf{reg}, \langle 1, 2\rangle)$ and $(\mathbf{reg}, \langle 3, 4\rangle)$ have different residuals wrt. $\mathcal{L}_{login}$. There is a distinct residual for every combination of $d_1, d_2 \in \mathcal{D}$ and $(\mathbf{reg}, \langle d_1, d_2\rangle)$. When comparing different data words, one has to abstract from concrete data values in residuals.

Definition 11 (Equivalence wrt. $\mathcal{L}_\mathcal{D}$) *Two words $u, u' \in \Sigma^*$ are equivalent wrt. $\equiv_\mathcal{L}$, denoted by $u \equiv_\mathcal{L} u'$, iff for some permutation π on $\mathcal{D}$*

$$\pi(\, u^{-1}\mathcal{L}_\mathcal{D}) \;=\; u'^{-1}\mathcal{L}_\mathcal{D}. \qquad \square$$

Intuitively, the permutations replace equality of residuals by some notion of isomorphism on residuals. This corresponds exactly to the role of registers in DRAs. In our example $(\mathbf{reg}, \langle 1, 2\rangle)$ and $(\mathbf{reg}, \langle 3, 4\rangle)$ are equivalent wrt. $\equiv_\mathcal{L}$. Using the permutation on $\mathcal{D}$ that maps 1 to 3 and 2 to 4 and is the identity otherwise, $(\mathbf{reg}, \langle 1, 2\rangle)$ becomes $(\mathbf{reg}, \langle 3, 4\rangle)$, and the residual of $(\mathbf{reg}, \langle 1, 2\rangle)$, i.e., the singleton set containing $(\mathbf{login}, \langle 1, 2\rangle)$, becomes equal to the residual of $(\mathbf{reg}, \langle 3, 4\rangle)$.

Identifying transitions. The transitions of a register automaton are guarded by logic formulas over binary (in-)equalities between registers and symbolic parameters. Similar to the case of AAAR, the guards can be understood as an abstraction on the (infinite) set of all inputs. While in the case of AAAR the abstraction is formulated globally on the set of inputs, here representative data inputs have to be chosen for every prefix of the language (i.e., in every location of an RA).[3]

Intuitively, such a set has to contain one data word for every path (sequence of transitions) in a canonical DRA. A representative data word for a path has to be capable of representing all other data words for that particular path. It will thus be a data word with just enough equal data values for this sequence of transitions. Such a word can represent all data words

[3] AAAR can also be extended to work with local sets of representative inputs. Such an extension is presented in [56].

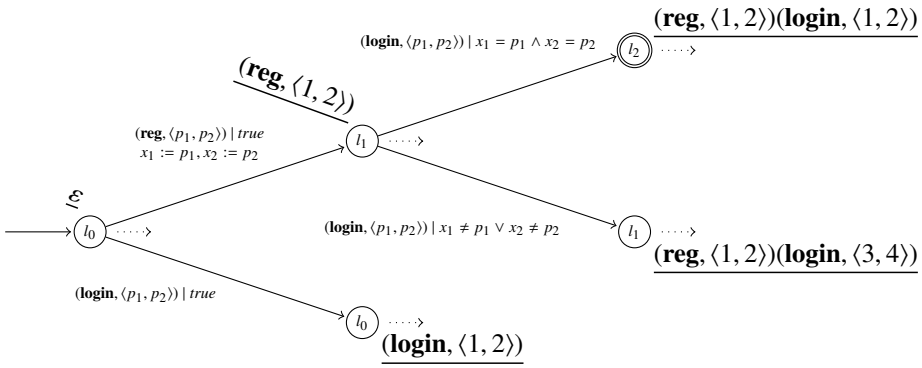

Figure 4.5: Prefix-closed subset of $\mathcal{L}$-essential words for $\mathcal{L}_{login}$

that correspond to the same path but have some additional equalities among memorable data values since the canonical DRA is right-invariant, i.e., equality of values in registers has no effect on its behavior. Let such words be $\mathcal{L}$-*essential* data words.

Figure 4.5 details the intuition for the example of $\mathcal{L}_{login}$ and its canonical DRA in Figure 4.4. The figure shows the prefix of the tree that results from "unrolling" the automaton from Figure 4.4 up to infinite depth. Every node in such a tree can be associated with one $\mathcal{L}$-*essential* data word form a prefix-closed set of $\mathcal{L}$-*essential* data words. The root of the tree corresponds to the empty data word (no transition has been passed so far).

The two nodes in the right of the figure and the corresponding $\mathcal{L}$-*essential* data words show what is meant by "just enough equal data values". The data word $(\mathbf{reg}, \langle 1, 2\rangle)(\mathbf{reg}, \langle 3, 4\rangle)$ is $\mathcal{L}$-essential since it cannot be the special case of any other word (it has no equal data values). This reflects that the guards of both transitions passed by this word do not require any equalities between data values. The word $(\mathbf{reg}, \langle 1, 2\rangle)(\mathbf{reg}, \langle 1, 2\rangle)$ is $\mathcal{L}$-essential since it is the word with least equal data values that is not a special case of $(\mathbf{reg}, \langle 1, 2\rangle)(\mathbf{reg}, \langle 3, 4\rangle)$, i.e., corresponds to a different sequence of transitions in the canonical DRA for $\mathcal{L}_{login}$, requiring exactly the first and third data value and the second and fourth data value to equal.

In [23] (Paper IV), we prove the existence of a unique minimal set of $\mathcal{L}$-essential words inductively, delivering implicitly an algorithm for constructing such a set. Of course, the definition in the paper is not based on the intuition given above, which relies on the (unknown) canonical DRA for a data language. It is based on an a set of orders on data words, realizing the intuition of special cases given above. Using these orders, $\mathcal{L}$-essential data words can be identified by means of suffixes. Intuitively, data words that can be represented by one $\mathcal{L}$-essential word have "similar" residuals. These residuals can be computed from the residual of the $\mathcal{L}$-essential word.

Identifying registers. Finally, a method is needed for identifying which data values to store in registers when constructing a canonical register automaton for some data language. Let $\pi_{d,d'} : \mathcal{D} \to \mathcal{D}$ be a transposition of d and d', i.e., mapping d to d' and vice versa, and the identity otherwise.

Definition 12 (Memorable data values) *For some word $u \in \mathcal{W}_{\mathcal{D}}$ a data value $d \in ValSet(u)$ is* memorable *(wrt. $\mathcal{L}_{\mathcal{D}}$) if for some suffix $v \in \mathcal{W}_{\mathcal{D}}$ and some transposition $\pi_{d,d'}$, where $d' \in \mathcal{D} \setminus Vals(uv)$,*

$$uv \in \mathcal{L}_{\mathcal{D}} \;\Leftrightarrow\; u\pi_{d,d'}(v) \notin \mathcal{L}_{\mathcal{D}}.$$

The set of memorable data values in u is denoted by $mem_{\mathcal{L}}(u)$. □[4]

Memorable data values have to be stored in registers of a DRA for $\mathcal{L}_{\mathcal{D}}$. For $(\mathbf{reg}, \langle 1, 2\rangle)$ both data values are memorable, which is proven for data value 1 by suffixes $\langle \mathbf{login}, \langle 1, 2\rangle\rangle$, and $\langle \mathbf{login}, \langle 3, 2\rangle\rangle$.

Using the above ideas, a Myhill/Nerode-like theorem for (regular) data languages and register automata can be established, stating that a data language can be represented by a register automaton if $\equiv_{\mathcal{L}}$ has a finite index. One direction of the proof of this theorem is an algorithm for constructing a register automaton for a regular data language from its set of $\mathcal{L}$-essential words, resembling the approach from Section 2.1.

4.3 Inferring register automata models

In order to infer register automata models, two problems remain to be solved. First, a finite set of suffixes has to incrementally approximate and finally realize the Nerode-relation on the (finite) set of prefixes, used to determine the locations and transitions of the inferred model. In particular, these suffixes have to be capable of identifying locations and registers. Also, these suffixes have to be equally applicable to different prefixes, which due to data values is not trivial.

Second, counterexamples have to be exploited in a way that guarantees strictly monotone progress towards the canonical DRA for an inferred data language. This section focuses on how the idea of an abstract representation, applied to sets of suffixes, leads to a finite set of suffixes with the desired properties, and how the ideas for handling counterexamples from the previous sections can be extended in the context of register automata. Put together, this results in an active learning algorithm for register automata models.

Presentation in this section served as a basis for in [46] (Paper VI) and is thus very similar to the paper.

4.3.1 Abstract suffixes

Approximating the Nerode-equivalence for data languages by means of a finite set of suffixes is different from the classic case (of inferring DFA or Mealy machine models) in a number of respects. While in the classic case a suffix is equally meaningful after any prefix, this is not true when learning register automata. Here, the data values in a suffix, in particular if they equal memorable data values in a prefix, have an impact that is specific to a prefix. Also, in the classic case suffixes are only used to distinguish states. When learning

[4] The definition used here is not the one from [23] (Paper IV), which relies heavily on the formalism used in the paper. The definition of memorable data values given here resembles the definition from [11]. While both definitions provide the same set of memorable data values for a prefix, the one used here shows better how suffixes can be used to identify memorable data values.

register automata, however, suffixes are additionally used to identify memorable data values in prefixes. Also, Definition 11 uses residuals, i.e., infinite sets of suffixes. It is not immediately obvious that any finite set of suffixes can realize the Nerode-relation in the case of data languages. This section discusses how these problems are solved in [49] (Paper V).

Assume an infinite data domain $\mathcal{D}_V$, disjoint from $\mathcal{D}$, which will be used in suffixes. Elements of $\mathcal{D}_V$ will be marked by an overline. An *abstract suffix* is a data word with data values in $\mathcal{D}_V$ and $\mathcal{D}$. Let by convention the data values from $\mathcal{D}_V$ always occur in fixed order $\bar{1}, \bar{2}, \ldots \bar{k}$ in an abstract suffix. For a prefix u, an abstract suffix v represents all concrete suffixes that can be obtained by some injective mapping from $\mathcal{D}_V$ to $\mathcal{D} \setminus (ValSet(u) \cup ValSet(v))$. Since data languages are closed under permutations on $\mathcal{D}$, all these suffixes lead to the same behavior after u. Thus, abstract suffixes can be used just like concrete suffixes. Consider the example of $\mathcal{L}_{login}$. For the prefix $(\mathbf{reg}, \langle 1, 2\rangle)$, the abstract suffix $(\mathbf{login}, \langle \bar{1}, \bar{2}\rangle)$ represents all concrete suffixes that can be obtained by an injective mapping from $\mathcal{D}_V$ to $\mathcal{D} \setminus \{1, 2\}$, e.g., $(\mathbf{login}, \langle 3, 4\rangle)$ or $(\mathbf{login}, \langle 5, 7\rangle)$.

Now, let $Z = \{z_1, \ldots, z_k\}$ be a finite set of *placeholders*, which is disjoint from $\mathcal{D}$ and $\mathcal{D}_V$. An *abstract parameterized suffix* $\langle v\rangle$ is a data word with data values in $Z \cup \mathcal{D}_V$. Let $\sigma : Z \to ValSet(u) \cup (\mathcal{D}_V \setminus ValSet(\langle v\rangle))$ be an injective mapping that is applied to $\langle v\rangle$ pointwisely to all $z_i \in Vals(\langle v\rangle)$ such that $\sigma(\langle v\rangle)$ is an abstract suffix. For a set V of abstract parameterized suffixes let $V(u)$ be the set of abstract suffixes that can be generated from abstract parameterized suffixes in V by mappings σ for prefix u.

Different mappings σ will use selected data values from $ValSet(u)$ in suffixes. They allow exactly the construction of pairs of suffixes that identify memorable data values. In the case of $\mathcal{L}_{login}$, consider the prefix $(\mathbf{reg}, \langle 1, 2\rangle)$. The abstract parameterized suffix $(\mathbf{login}, \langle z_1, z_2\rangle)$ will (among others) yield the abstract suffixes $(\mathbf{login}, \langle 1, 2\rangle)$ and $(\mathbf{login}, \langle \bar{1}, 2\rangle)$, which prove 1 to be memorable in $(\mathbf{reg}, \langle 1, 2\rangle)$.

While abstract parameterized suffixes solve the problem of identifying memorable data values, they introduce a new problem. Prefixes of different length (i.e, different numbers of data values) give rise to differently many abstract suffixes for one abstract parameterized suffix. In order to use abstract suffixes for approximating the Nerode-relation, however, the partial residuals constructed by means of abstract parametrized suffixes have to be related for different prefixes. The sets of abstract suffixes have to be represented in a way that allows comparing. Luckily, the idea of an (abstract) representation, used in Section 4.1 and Section 4.2.2 on the (infinite) set of inputs and the set of prefixes, respectively, can be adapted to sets of suffixes.

For two abstract suffixes v, v'in $V(u)$ let $v \sqsubseteq_V v'$ denote that v' can be obtained from v by an injective mapping from $\mathcal{D}_V$ to $\mathcal{D} \cup \mathcal{D}_V$. In the example used above $(\mathbf{login}, \langle \bar{1}, \bar{2}\rangle) \sqsubseteq_V (\mathbf{login}, \langle 1, \bar{1}\rangle) \sqsubseteq_V (\mathbf{login}, \langle 1, 2\rangle)$ for prefix $(\mathbf{reg}, \langle 1, 2\rangle)$. The preorder $\sqsubseteq_V$ can be used to define a set of representative suffixes for $V(u)$. For $v \in V(u)$ let

$$weak_{V(u)}(v) =_{def} max_{\sqsubseteq_V}\{v' \in V(u) \mid v' \sqsubseteq_V v\},$$

i.e., the set of greatest (wrt. $\sqsubseteq_V$) suffixes in the set of suffixes smaller than v (wrt. $\sqsubseteq_V$) in $V(u)$.

A suffix v is in the set of *representative suffixes* for $V(u)$ if for some v' in $weak_{V(u)}(v)$ it is true that w.l.o.g. $uv \in \mathcal{L}_{\mathcal{D}}$ but $uv' \notin \mathcal{L}_{\mathcal{D}}$, or if $weak_{V(u)}(v)$ is empty. Intuitively, the

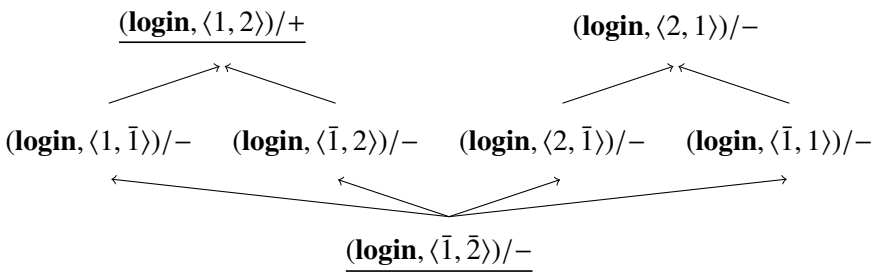

Figure 4.6: Preorder of abstract suffixes and corresponding acceptance for prefix (**reg**, ⟨1, 2⟩) generated by abstract parameterized suffix (**login**, ⟨z_1, z_2⟩) in the case of $\mathcal{L}_{login}$. Representative suffixes marked by an underline

set of representative suffixes contains all suffixes in which a memorable data value shows some effect and additionally the smallest suffixes wrt. $\sqsubseteq_V$. This representation allows for comparing sets of suffixes since it depends on the memorable data values of a prefix and not on all data values.

Figure 4.6 shows the construction of the set of representative suffixes for prefix (**reg**, ⟨1, 2⟩) and abstract parameterized suffix (**login**, ⟨z_1, z_2⟩) in the $\mathcal{L}_{login}$ example. Suffixes in the preorder are marked with + if the corresponding word (concatenated to the prefix) is in $\mathcal{L}_{login}$, and with − otherwise. Representative suffixes are underlined. As can be seen, the smallest element and the suffix identifying the memorable data values in (**reg**, ⟨1, 2⟩) are representative suffixes.

For a set of abstract parameterized suffixes V and some prefix u, let $\lfloor V(u) \rfloor$ be the set of representative suffixes for $V(u)$, and the *closure* $C_{V(u)} : \lfloor V(u) \rfloor \to \{+, -\}$, where $C_{V(u)}(v) = +$ if $uv \in \mathcal{L}_{\mathcal{D}}$ and − otherwise for v in $\lfloor V(u) \rfloor$. Let further $mem_V(u)$ be the set of data values in $ValSet(u)$ that are proven to be memorable in u by V. For two prefixes $u \equiv_{\mathcal{L}} u'$ obviously $|mem_V(u)|$ equals $|mem_V(u')|$, and there is a permutation π on $\mathcal{D}$ that on $mem_V(u)$ is a bijection into $mem_V(u')$ such that $\pi(C_{V(u)}) = C_{V(u')}$, where π is applied to all suffixes in the domain of $C_{V(u)}$.

So far, it has been shown how abstract suffixes can be used to realize the Nerode-relation on a set of prefixes and how abstract suffixes identify memorable data values of these prefixes. It remains to be shown that a finite set of abstract suffixes can realize the Nerode-relation such that $\pi(C_{V(u)}) \neq C_{V(u')}$ for all permutations π on $\mathcal{D}$ in case $u \not\equiv_{\mathcal{L}} u'$. It follows from Definition 11 that if $u \not\equiv_{\mathcal{L}} u'$, then for every permutation π there is at least one suffix v such that $uv \in \mathcal{L}_{\mathcal{D}} \Leftrightarrow u'\pi(v) \notin \mathcal{L}_{\mathcal{D}}$. Thus, it only has to be shown that a finite number of suffixes is sufficient to eliminate all permutations in case $u \not\equiv_{\mathcal{L}} u'$

For $u, u' \in \mathcal{W}_{\mathcal{D}}$, let there be more than one permutation on $\mathcal{D}$ under which the partial representative residuals become equal for some set V of abstract suffixes, i.e.,

$$\pi_1(C_{V(u)}) = C_{V(u')} = \pi_2(C_{V(u)}).$$

This implies that $C_{V(u)} = \pi_1 \circ \pi_2^{-1}(C_{V(u)})$. Either there are memorable data values in u that can be used interchangeably in $\mathcal{L}_{\mathcal{D}}$, or the suffixes in V fail to prove these data values to be non-interchangeable. In the latter case a single abstract suffix can be found, splitting a

group of seemingly interchangeable memorable data values in $C_{V(u)}$. Such a suffix has to prove two memorable data values of u to have different effects. It is sufficient to find a suffix in which the one data value has an effect while the other does not. Since the two memorable data values can be distinguished in $\mathcal{L}_\mathcal{D}$, such a suffix exists.

Now, let V be such that for $u \not\equiv_\mathcal{L} u'$ all memorable data values are identified and only true symmetries remain between memorable data values, but still $\pi(C_{V(u)}) = C_{V(u')}$ for some permutation π on $\mathcal{D}$. As stated above, there is a single suffix that disproves π. It is sufficient to argue about a single permutation here. If permutations π_1, π_2 remain, the complete residual of u is invariant under $\pi_1 \circ \pi_2^{-1}$. Disproving one permutation, automatically disproves the other one, too.

Hence, a finite set of abstract parameterized suffixes can be used to realize the Nerode-relation on a set of prefixes and to identify registers in a register automaton to be constructed.

4.3.2 An active learning algorithm for register automata

The ideas from the previous sections can be combined into the active learning algorithm for register automata models presented in [49] (Paper V). The algorithm is formulated in the MAT learning model, using membership queries for data words and equivalence queries that test the equivalence of an unknown target data language $\mathcal{L}_\mathcal{D}$ and the data language recognized by an intermediate hypothesis.

The learning algorithm will use a finite prefix-closed set of $\mathcal{L}$-essential words as prefixes. The prefixes will be used to determine locations and transitions of the inferred register automaton. The Nerode-relation on the set of prefixes will be approximated by a finite set of abstract parametrized suffixes. As a data structure an observation table, i.e., a mapping from the set of prefixes to closures for prefixes and abstract parameterized suffixes, is used. According to the Nerode-relation for data languages, prefixes in the table are compatible if their closures become equal under some permutation π on $\mathcal{D}$. As in the classic case, the learning algorithm will start with the empty data word as only prefix and only suffix. The empty prefix leads to the initial location in the canonical DRA for $\mathcal{L}_\mathcal{D}$, and the empty suffix distinguishes prefixes in $\mathcal{L}_\mathcal{D}$ from ones that are not in $\mathcal{L}_\mathcal{D}$.

Extending the ideas from Section 4.1, counterexamples are analyzed in order to determine if they (1) prove a new data value to be memorable in some prefix, (2) disprove a permutation under which the approximated residuals of two prefixes become equal, or (3) prove a refinement of the abstraction to be necessary (i.e., provide a new $\mathcal{L}$-essential prefix).

Figure 4.7 shows this extended analysis of counterexamples. The first case (new location) resembles the default case from Theorem 2: data words $\lfloor u \rfloor_\mathcal{H} a$ and $\lfloor ua \rfloor_\mathcal{H}$ from the set of prefixes lead to the same location in the hypothesis but the suffix of the counterexample disproves their equivalence (at least under the permutation used during hypothesis construction). Adding a new abstract parameterized suffix generated from the suffix of the counterexample to the set of suffixes used by the learning algorithm will reduce the number of valid permutations between both prefixes as discussed in Section 4.3.1.

The second case (new transition) corresponds to the case of a refinement in AAAR stated in Theorem 4: The data word ua' is the $\mathcal{L}$-essential word for ua in the hypothesis. In case this is true in the SUL, too, a counterexample can be found in which the prefix ua is replaced by its $\mathcal{L}$-essential representative word ua'. The search for such a counterexample

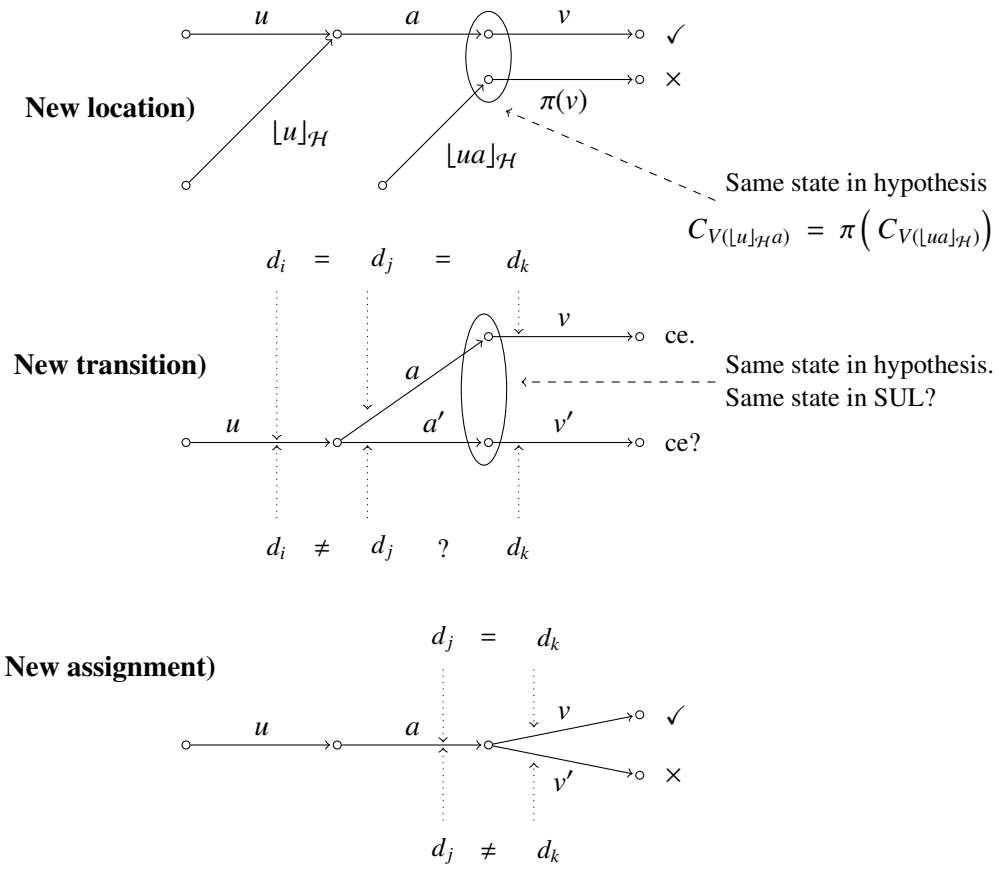

Figure 4.7: Handling counterexamples when inferring register automata models

can be quite expensive. As indicated in the figure, the new counterexample will have less equalities between data values as the original one. The number of possible refinements is exponential in the number of equal data values that have to be refined into multiple sets of equal data values.

The third case (new assignment) relates directly to the definition of memorable data values (Definition 12). The suffixes v and v' are a pair of suffixes as assumed in the definition. The word ua is from the set of prefixes. This ensures that the newly introduced assignment has a well-defined transition in the next hypothesis.

Progress achieved in any of the three dimensions is strictly monotonic. Due to the finite index of $\equiv_{\mathcal{L}}$ there are only finitely many locations and registers in the canonical DRA for $\mathcal{L}_{\mathcal{D}}$, and only finitely many transitions that have to represented by $\mathcal{L}$-essential words. As shown in Section 4.3.1, the number of permutations between two prefixes can only be reduced finitely often before no valid permutation is left. Thus, the algorithm will terminate with the unique canonical DRA for $\mathcal{L}_{\mathcal{D}}$ (up to isomorphism, i.e, renaming of locations and registers).

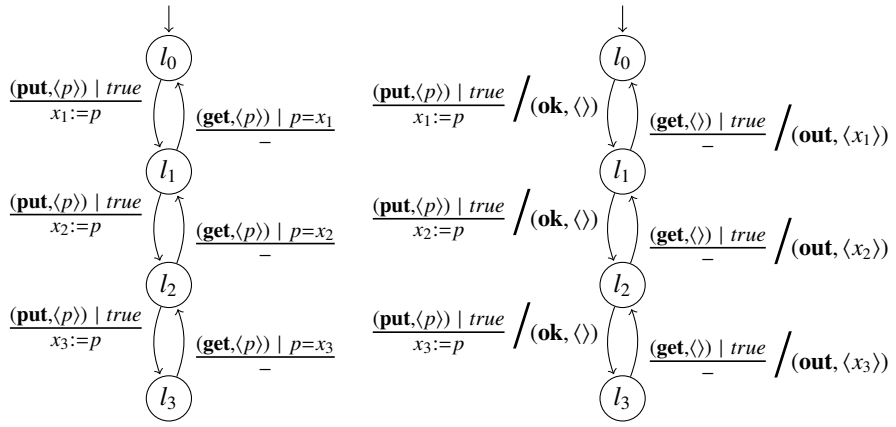

Figure 4.8: Partial models of a stack of capacity 3: RA (left) and RMM (right)

4.4 Inferring semantic interfaces of data structures

The previous sections summarized the ideas for extending active learning to register automata. While register automata accept data languages, real black-box components are usually reactive systems, producing output. In [46] (Paper VI) an extension of the algorithm from [49] (Paper V) to systems that produce outputs with data values from registers is used to infer interfaces of data structures. Rather than re-describing the extension in detail here, this section will only provide the intuition behind the extension and discuss its effect along a small example.

As an example for a system that produces parameterized output think of a stack of limited size, exposing two inputs: **put** and **get**. The action **put** tries to store an additional object in a data structure, **get** tries to retrieve a data value from a data structure. The stack allows storing the same object multiple times, and does not change its state on unsuccessful operations, i.e., when adding an additional data value exceeding the size limit of the stack, or when trying to get a data value from the empty stack.

The stack can be modeled as a register automaton, which is shown in the left-hand side of Figure 4.8. For better readability reflexive transitions are omitted in the figure. The **get** operation in this case has a data parameter and the returned data value of get operations is encoded in the guards of corresponding transitions. E.g., from location l_1, a **get** returns the value of register x_1. In terms of the data language accepted by the RA, the data word $(\mathbf{put}, \langle d \rangle)(\mathbf{get}, \langle d \rangle)$ is in the data language accepted by the automaton.

While it is certainly possible to model parameterized output in this fashion, it is not very natural. In the right-hand side of Figure 4.8 the same stack is modeled as a Register Mealy Machine (RMM), i.e., a register automaton with Mealy machine-like outputs on all transitions [46] (Paper VI). There, the outputs are symbolic: they comprise data parameters, which are bound to registers. Performing a **get** operation from location l_1 will produce an output $(\mathbf{out}, \langle x_1 \rangle)$.

In order to extend active learning from register automata to RMMs, the concepts introduced in the previous sections have to be modified accordingly.

Table 4.1: Experimental results for inferring register automata models from data structures using various algorithms

Name	Mealy ($\lvert\mathcal{D}\rvert = 4$)			RA			RMM		
	$\lvert Q\rvert$	MQs	EQs	$\lvert L\rvert$	MQs	EQs	$\lvert L\rvert$	MQs	EQs
Stack (1)	5	17	0	3	35	2	2	10	0
Stack (2)	21	53	1	4	135	4	3	18	0
Stack (3)	85	232	3	5	554	6	4	38	1
Stack (4)	341	854	4	6	2998	8	5	53	2

1. The Nerode-relation for data languages has to be adapted in a fashion similar to the semantic functionals of Mealy machines from Section 2.1.

2. The definition of memorable data values has to be extended by the additional case that a data value is proven to be memorable by occurring in some output.

3. Closures and their equivalence have to be adapted to residual functionals with parameterized outputs.

Apart from these extensions, the algorithm from [49] (Paper V) can mostly remain unchanged. Corresponding to the extension of active learning from DFAs to Mealy machines, the set of abstract suffixes will be initialized using all symbolic inputs instead of the empty word.

Implementations of both learning algorithms (for register automata and RMMs) have been evaluated in a small series of experiments based on models of data structures. The learning algorithms have been implemented on top of LearnLib. The results from all conducted experiments are presented in [46] (Paper VI). Here, the results from one series of experiments will be discussed. In this series, models of stacks with varying size limits have been inferred using three different approaches.

1. Mealy machine models have been learned for a finite data domain of size 4. In these experiments, the learning algorithm for Mealy machines was equipped with a technique for symmetry reduction, resulting in optimized consumption of membership- and equivalence queries.

2. Register automata models have been inferred using the learning algorithm from [49] (Paper V).

3. The active learning algorithm from [46] (Paper VI) has been used to infer for RMM models.

In all experiments counterexamples were derived by "unfolding" the target system and hypothesis models for a large enough finite data domain, and applying an equivalence test for Mealy machines to these models. Table 4.1 shows the key figures of the experimental results.

While the inferred Mealy machine models encode the state of the stack in the state space, the inferred register automata and RMM models have one location per number of stored

data values, including an initial (empty) location. The register automata additionally have one sink location, which is an artifact of the instrumentation used in this setup.

In the series of experiments inferring Mealy machines and register automata, the number of membership queries grows exponentially in the size of the stack. In the case of Mealy machines this is due to the exponentially growing state-space.

In the the case of register automata this has two reasons. First, as described in the analysis in [49] (Paper V), the number of registers contributes to the number of membership queries as an exponential factor. Second, the register automaton encodes output into guarded transitions. These additional guarded transitions have to be inferred from counterexamples.

As discussed in [49] (Paper V), deriving new prefixes representing guarded transitions from counterexamples requires exponentially many membership queries (in the number of data values in a counterexample). In the series of experiments presented here, the costs of inferring additional guards clearly dominate the costs for inferring registers, which is indicated by the increased number of equivalence queries in the case of inferring register automata (compared to the experiments with the algorithm for learning RMMs).

The experiments lead to two conclusions. First, switching from Mealy machines to models with registers leads to more expressive models: Modeling the influence of (stored) data values on the control-behavior results in infinite-state models. Already for small finite data domains these infinite-state models are exponentially smaller than the corresponding Mealy machines.

Second, using RMMs instead of register automata does not further extend the expressiveness of the inferred models but has a dramatic effect on the performance. The more adequate way of modeling output leads to exponential (in the number of registers and length of counterexamples) savings in the number of membership queries. The number of equivalence queries is reduced roughly by the number of transitions with parameterized output.

The huge impact of inferring RMMs instead of Mealy machines becomes apparent when looking at bigger systems. The RMM model of a nested stack (i.e., a stack of stacks) with a capacity of four in the inner stacks as well as in the outer stack has 781 locations, and can be inferred with 44,589 membership queries and only 9 equivalence queries (cf. [46] (Paper VI)). The RMM has 16 registers, corresponding to the overall capacity of the 2-dimensional stack. The corresponding Mealy machine model for a finite data domain of size four would already have significantly more than 4^{16} ($> 10^9$) states, which clearly exceeds the capabilities of today's active learning algorithms by far.

5 Related work

This chapter reviews works related to the topic of this thesis, i.e., the inference of interface programs by means of (active) automata learning. The structure of the chapter follows roughly the topics addressed in the previous chapters.

Active automata learning. Automata learning techniques have been researched for the past 40 years. At first passive learning algorithms, constructing automata from gives sets of examples have been investigated (e.g., [16, 84]). In [37] it is shown that constructing minimal automata from samples is an NP-complete problem.

The first active automata learning algorithm, using membership queries and a set of prefixes that covers all classes of the Nerode-relation, is presented in [6]. The algorithm incorporates the partition-refinement pattern typical to active learning and uses a polynomial number of membership queries.

Active learning for regular languages with membership queries and equivalence queries is presented in [7]. Variants are presented in [60], using a discrimination tree instead of an observation table, and in [62], adding suffixes of counterexamples to the table instead of prefixes. A first algorithm for analyzing counterexamples explicitly is presented in [75]. The abstract idea of an observation pack, unifying these approaches is developed in [8].

Active learning has been extended to Mealy machine models in [70, 63] by adapting the algorithm from [7]. In [77] the algorithm from [62] is adapted in a slightly optimized version to Mealy machine models. Further improvements to this approach, especially for the case of long counterexamples obtained from testing or monitoring, are discussed and evaluated in [54]. Finally, in [82] (Paper I), the strategy from [75] for handling counterexamples explicitly is extended to Mealy machines.

The performance of active learning (in terms of queries) is essential when it comes to practical application. This has been addressed by applying application specific filters, which help answering membership queries without performing tests on a SUL [53, 9]. The potential of different generic types of filters and the interplay of different filters has been investigated in [65, 64, 55]. A different approach to dealing with realistic systems is the inference of partial models as is discussed in Section 3.1.

While optimizations to the number of membership queries have been investigated thoroughly, equivalence queries remain a fundamental problem in active learning. They do not exist in practice but can only be approximated using methods from testing. In [12] the close relation between active automata learning and conformance testing (e.g., [24, 33]) is analyzed. An example for the integration of active learning and conformance testing is given is [72], where the combined approach is discussed as one method for checking properties of black-box systems.

Applications of active learning. Currently there are two libraries available implementing active learning algorithms, namely libalf [20] and LearnLib [74, 67]. While the focus of libalf is providing implementations of algorithms from the literature, LearnLib aims at providing an infrastructure for evaluating, and implementing learning algorithms as well as for applying these in case studies. The implementations of (active) learning algorithms from libalf can be used in the framework provided by LearnLib.

Active learning has been applied successfully in an number of interesting case studies. It has been used to infer models of CTI systems [41, 40], web-applications [73], communication protocol entities [1], the new biometric European passport [2], bot nets [21], a network of integrated controllers in the door of a car [80], and enterprise applications [10, 9]. The particular challenges of practical application are discussed in [48] along with illustrating examples from case studies.

The state-of-the-art approach to inferring models of real (i.e., infinite state) systems in active automata learning is using predefined abstractions, realized by so-called *mappers* (cf. [58]), which are placed between the component to be inferred and the learning algorithm. Mappers provide a regular projection of the component. This approach is described explicitly for the generation of specifications from protocol entities in [1].

However, defining mappers is an error-prone and laborious manual effort. In [51] (Paper III) automated alphabet abstraction refinement is integrated with active learning to overcome this problem. A similar approach, using lazy alphabet construction without counterexample driven alphabet abstraction refinement, is presented in [35] for the inference of assumptions in assume-guarantee reasoning.

Extensions to active learning. A different approach to dealing with non-regular systems is the extension of active learning algorithms to richer modes. Active automata learning has been extended to systems with parameterized inputs and guarded transitions in a number of works [45, 79, 78, 13]. Other extensions include I/O-automata [3] (using a mapper that provides a regular projection of the inferred I/O-automaton), timed automata [38, 39] (using decision trees similar to the ones presented in [23] (Paper IV)), Petri Nets [30], or Kripke-structures [66]. An extension to probabilistic systems is presented in [31].

The requirements on modeling interface programs have been analyzed in [47]. The resulting automaton model (i.e., register automata) is presented in [23] (Paper IV). It resembles *Memory Automata*, which have been introduced in [59] with the explicit aim of developing an automata-based formalism for program behavior. Register automata differ from memory automata mainly in the explicit description of guards and assignments along transitions and by using a finite set of inputs with data parameters instead of using an infinite set of inputs directly. Myhill/Nerode-like theorems for classes of Memory Automata have been formulated in [11] and in [32]. The canonical models resulting from the Myhill/Nerode-like theorem presented in [23] (Paper IV) differ significantly from canonical Memory Automata. In fact, by allowing random access to registers canonical RAs can be exponentially smaller than the canonical Memory Automata from [11] and [32]. A detailed discussion can be found in [23] (Paper IV). In [18, 19], automata with registers are investigated from an algebraic perspective, resulting in a Nerode-relation similar to the one of [23] (Paper VI).

An active learning algorithm for Memory Automata is presented in [76]. This algorithm infers DFAs for growing finite data domains and encodes these as Memory Automata. Un-

like the algorithm presented in [49] (Paper V), no guarantees about the resulting automata are made. In particular, since the approach is not backed by a canonical form, the automata constructed in different learning experiments (for the same SUL) may differ. The authors of [14, 17] present a similar technique for inferring *symbolic Mealy machines*, i.e., automata with guarded transitions and state-local sets of registers: First, a Mealy machine is learned for a large enough finite domain. Then, from this Mealy machine a symbolic version is constructed. In [61] another variant of this two-step approach is presented in combination with passive learning: behavioral models are inferred from program traces obtained through monitoring using passive automata learning. The influence of data values on the behavior is inferred with an invariance detector [29]. The results are combined into a single model.

In contrast to these two-step approaches that use off-the-shelf learning algorithms in the first step, the active learning algorithm for register automata presented in [49] (Paper V) infers models with registers directly. The extension of active learning to this class of richer models comes at the price of a more complex treatment of counterexamples and requires appropriate extensions of the suffix-based identification of entities in models (i.e., registers, transitions, and locations). As can be seen in the results from the experiments presented in Section 4.4 and in [46] (Paper VI), however, this effort is rewarded already when inferring small models by a significant positive impact on the number of required tests that have to be executed on a SUL.

Interface synthesis. Synthesis of component interfaces has been a research interest for the past decade. Presented approaches fall into three classes described in [81].

First, *client-side static analysis* uses a static analysis of source code using the component of which a model is to be inferred. The approaches described in [81, 86] mine Java code to infer sequences of method calls. These sequences are used to produce usage patterns statistically. Deviations from these patterns are then considered as candidates for erroneous usages of the component and have to be verified manually.

Second, *component-side static analysis* uses a static analysis on the component itself. In [4] an approach is presented that generates behavioral interface specifications for Java classes by means of predicate abstraction and active automata learning. Here, predicate abstraction is used to generate an abstract version of the considered class. Afterwards a minimal interface for this abstract version in obtained by active learning. Another approach uses counterexample guided abstraction refinement (CEGAR) [42] instead of active learning in order to derive a regular model from the Boolean program obtained by predicate abstraction. Both strategies are compared in [15].

Static analysis methods rely on access to source-code, either of the component or of code using the component. This cannot be assumed in all scenarios. The CONNECT project [57], which provides the motivational background for this thesis, aims at automated mediator synthesis for networked systems. In this scenario neither components are available as source-code, nor is client-side code.

The third class of methods, namely *dynamic analysis*, infers interface models from actual program executions. The authors of [5] present an approach for inferring probabilistic finite state automata (PFSA) describing a components' interface using a variant of the k-tail algorithm [16] for learning finite state automata from sets of examples. In [87] a special form of component interfaces, consisting of one finite state automaton per field of a class, is

presented along with a static analysis and a dynamic analysis for inferring such component interfaces.

Finally, the SPY approach [36] aims at inferring behavioral specifications, represented as graph transformation rules, from Java classes. Inference is organized in two phases: First, the run-time behavior of components is observed on a small concrete data domain. Then, (symbolic) transformation rules are constructed inductively from the concrete observations.

All three of these dynamic approaches use passive learning and are thus limited to (possibly small) sets of observed executions. In case some functionality of a component is not executed, it will not be captured in the inferred model. Using active methods, on the other hand, allows for active exploration of the interface of a component: The authors of [27], e.g., use a combination of component-side static analysis, identifying side-effect free methods (so-called *inspectors*), which are then used to identify states of the component when interacting with the component systematically.

Summarizing, all previous approaches that infer models describing data explicitly from black-box components (i.e., [36], [76], [14], and [61]) work in two steps. The first step consists in inferring models (or in obtaining sets of examples in the case of passive learning) at the level of a concrete finite data domain. The experiments from [46] (Paper VI) show that this becomes unfeasible already for relatively small models of data structures. The active learning algorithm presented in this thesis, on the other hand, infers register automata directly, without generating such an intermediate representation.

Except for [36], the approach to interface synthesis presented to in this thesis differs from all previous attempts wrt. the expressiveness of the inferred models. While in all other approaches inferred interfaces are modeled as “plain” automata, the automata used in this thesis allow for modeling of (restricted) interface programs comprising conditions and assignments.

6 Conclusion and future work

6.1 Conclusion

This thesis addresses the problem of developing active learning algorithms for interface programs describing interaction protocols of black-box systems capable of storing and comparing data values used in the interaction. As has been discussed in Section 1.1, this required research in three directions: First, the efficiency of active learning (in terms of membership queries) had to be optimized. Second, equivalence queries had to be approximated. Third, active learning had to be extended to richer models than Mealy machines or DFAs. The work presented in this thesis improves the state of the art in all of these directions:

- The efficiency (in terms of membership queries) of active learning for DFAs and Mealy machine models has been improved considerably by using a new method for handling counterexamples in combination with inferring partial models: A model with more than 166,000 states and 121 inputs was inferred. This exceeds the state of the art (22,000 states and 11 inputs) by more than two orders of magnitude (cf. Section 3.2.3).

- The incremental equivalence queries presented in Section 3.2 clearly outperformed the best other algorithms for approximating equivalence queries in the ZULU competition, discovering up to 8% more states using the same amount of membership queries (cf. [50] (Paper II)).

- Automated alphabet abstraction refinement allows for inferring models of real systems (with infinite sets of inputs) without an intermediate hand-tailored abstraction. An optimal abstraction is inferred in the course of learning (cf. Section 4.1). In the case of a biometric passport [2], e.g., a model and an abstraction was learned fully automatically. Designing the same abstraction required considerable manual effort in the original case study, presented in [2].

- The extension of active learning to register automata models, presented in Section 4.4, allows for the inference of (restricted) program structures comprising conditions and assignments, capable of operating real black-box components. Unlike previous approaches to the inference of models or interfaces with similar expressivity (e.g., [14] or [36]), no unfeasibly large finite state representation of the system under learning has to be produced. Instead, register automata are constructed from observations directly.

  The practical importance of this improvement becomes apparent when inferring RMMs (cf. Section 4.4). Already for small data domains finite state representations of the models inferred in the experiments in [46] (Paper VI) exceed the biggest Mealy

machine models inferred in experiments presented in this thesis (cf. Section 3.2.3) by several orders of magnitude.

In the particular case of a nested stack of overall capacity 16, the resulting Mealy machine model for a finite data domain of size 4 would have 7 inputs and significantly more than 10^9 states. The biggest inferred Mealy machine model presented in this thesis is more than two orders of magnitude smaller and was inferred with almost 50 million membership queries and more than 400,000 equivalence queries (cf. Table 3.4) in almost 3 hours. The RMM of the nested stack, on the other hand, was inferred with less than 45,000 membership queries and only 9 equivalence queries in no more than 20 seconds.

Key to progress in all directions are two related and recurring patterns, which are extended step-wisely throughout the thesis: The suffix-based identification of entities in inferred models and the corresponding analysis of counterexamples.

Suffix-based identification. When inferring Mealy machine models, only states are identified by means suffixes. In the case of automated alphabet abstraction refinement, the pattern is extended to classes of the alphabet abstraction, which are identified using tuples of prefixes and suffixes. Finally, in the case of inferring register automata, single (concrete) suffixes are no longer sufficient to identify entities in models: Locations and registers are identified using abstract parameterized suffixes, each representing a complete preordered set of concrete suffixes.

Analysis of counterexamples. Accordingly, the decomposition and analysis of counterexamples is extended to multiple cases of counterexamples.

1. The thorough analysis of possible strategies for handling counterexamples when inferring Mealy machine models from Section 2.2.2 provides a solid base case, i.e., counterexamples leading to new states, which can be used also in the more elaborate scenarios.

2. The second case, i.e., counterexamples to an abstraction and its integration with the base case, is presented in Section 4.1. It can be adapted to the case of counterexamples leading to new transitions in the case of inferring RAs.

3. Finally, a third case, i.e., counterexamples leading to new register assignments, is added "on top" of the previously introduced cases when inferring register automata models in Section 4.3.2.

The seamless integration of all three cases is achieved by the decomposition of counterexamples: In all cases counterexamples are decomposed into an access sequence to one of the states of a hypothesis and some suffix. This decomposition guarantees that all newly discovered locations, transitions, and registers have a well-defined, i.e., invariant, position in subsequent hypothesis models.

The combination of both patterns extends the typical partition-refinement based approach of active automata learning algorithms to multiple dimensions: locations, transitions, and registers.

6.2 Future work

The application of active automata learning in the context of interface synthesis leads to several interesting questions which exceed the scope of this thesis. These questions can be categorized using the categories from Section 1.1.

Performance in practical scenarios. The learning algorithm for register automata presented in [49] has a worst case complexity that is influenced exponentially by the length of counterexamples. This complexity had a strong negative impact on the performance in experiments where RA models were inferred of systems that produce output. One big step towards the practical application of active learning of interface programs was the extension to RMMs presented in Section 4.4. This practical optimization did not only lead to more natural models of reactive systems but also increased the performance of learning dramatically.

However, this optimizations came at a price: in the current version of RMMs newly introduced data values in outputs are not covered. In real systems this is a quite common phenomenon: As an example consider a session identifier that is created by the component to be inferred. In order to infer interface programs for a larger class of components efficiently, it will have to be investigated how inference of RMMs can be extended to handle output comprising new data values.

Equivalence queries. The approximation of equivalence queries still is one of the fundamental open problems in active automata learning. While the evolving hypothesis algorithm, presented in this thesis, has been applied successfully in the ZULU challenge, the systems used there were randomly generated. In this scenario randomized generation of suffixes proved to be the best strategy. This strategy, however, will hardly be successful for man made systems.

On the other hand, a number of works explore the combination of white-box and black-box methods (or static analysis and dynamic analysis) for the synthesis of interfaces. It would be very interesting to investigate if equivalence queries, i.e., the search for counterexamples, can benefit from information obtained by static analysis (e.g., symbolic execution), or monitoring.

Expressivity. Register automata faithfully describe systems that only pass data and do not process data. While this is sufficient for modeling an interesting class of systems (e.g., data structures and many network protocols), it does not include all networked systems envisioned in the CONNECT project: More complex components and protocols use data parameters on which simple operations can be performed, e.g., an increment operation on a sequence number.

One appealing direction of research is the extension of active learning to more expressive models, allowing more tests on data values than for equality, or allowing simple operations on data values. Interesting phenomena of practical importance can be derived, e.g., from the requirements in the context of the CONNECT project. In particular, the boundary will have to be explored up to which suffix-based identification can be used to uniquely identify these phenomena in a black-box scenario.

Bibliography

[1] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. In *Proceedings of the 22th IFIP WG 6.1 Int. Conf. on Testing software and systems, ICTSS'10*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer Verlag, 2010.

[2] Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and Abstraction of the Biometric Passport. In *Proceedings of the 4th Int. Symposium on Leveraging Applications, ISoLA'10 (1)*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer Verlag, 2010.

[3] Fides Aarts and Frits Vaandrager. Learning I/O Automata. In Paul Gastin and François Laroussinie, editors, *Proceedings of the 21th Int. Conf. on Concurrency Theory, CONCUR'10*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer Verlag, 2010.

[4] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05*, pages 98–109. ACM, 2005.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05*, pages 4–16. ACM, 2002.

[6] Dana Angluin. A Note on the Number of Queries Needed to Identify Regular Languages. *Information and Control*, 51(1):76–87, 1981.

[7] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[8] José L. Balcázar, Josep Díaz, and Ricard Gavaldà. Algorithms for Learning Finite Automata from Queries: A Unified View. In D.-Z. Du and K.-I. Ko, editors, *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Kluwer Academic, 1997.

[9] Oliver Bauer. Beherrschung emergenten Verhaltens auf Basis regulärer Extrapolation am Beispiel einer prozessgesteuerten Anwendung. Master's thesis, TU Dortmund, Department of Computer Science, Chair of Programming systems, 2011.

[10] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing System States by Active Learning Algorithms. In *Proceedings of the 1th EternalS workshop, EternalS'11*, volume 255 of *Communications in Computer and Information Science*, pages 61–78. Springer Verlag, 2012.

[11] M. Benedikt, C. Ley, and G. Puppis. What You Must Remember When Processing Data Words. In *Proceedings of the* 4^{th} *Alberto Mendelzon Int. Workshop on Foundations of Data Management*, volume 619 of *CEUR Workshop Proceedings*.

[12] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In *Proceedings of the* 8^{th} *Int. Conf. on Fundamental Approaches to Software Engineering, FASE '05*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, 2005.

[13] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines with Parameters. In *Proceedings of the* 9^{th} *Int. Conf. on Fundamental Approaches to Software Engineering, FASE '06*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 2006.

[14] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *Proceedings of the* 11^{th} *Int. Conf. on Fundamental Approaches to Software Engineering, FASE '08*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer Verlag, 2008.

[15] Dirk Beyer, Thomas Henzinger, and Vasu Singh. Algorithms for Interface Synthesis. In *Proceedings of the* 19^{th} *Int. Conf. on Computer Aided Verification, CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 4–19. Springer Verlag, 2007.

[16] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.*, 21:592–597, June 1972.

[17] Therese Bohlin and Bengt Jonsson. Regular Inference for Communication Protocol Entities. Technical report, Department of Information Technology, Uppsala University, Schweden, 2009.

[18] Mikolaj Bojanczyk. Data Monoids. In *Proceedings of the* 28^{th} *Int. Symposium on Theoretical Aspects of Computer Science, STACS'11*, volume 9, pages 105–116. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[19] Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata with Group Actions. In *Proceedings of the* 26^{th} *Annual IEEE Symposium on Logic in Computer Science, LICS'11*, pages 355–364. IEEE Computer Society, 2011.

[20] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The Automata Learning Framework. In *Proceedings of the* 22^{th} *Int. Conf. on Computer Aided Verification, CAV'10*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer Verlag, 2010.

[21] G. Bossert, G. Hiet, and T. Henin. Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems. In *Proceedings of the* 6^{th} *Conf. on Network and Information Systems Security, SAR-SSI'11*, pages 1–8. IEEE Computer Society, 2011.

[22] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[23] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A Succinct Canonical Register Automaton Model. In *Proceedings of the* 9^{th} *Int. Symposium on Automated Technology for Verification and Analysis, ATVA'11*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer Verlag, 2011.

[24] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[25] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of the* 9^{th} *Int. Conf. on Tools and algorithms for the construction and analysis of systems, TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.

[26] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: an Interactive Learning Competition. In *Finite-State Methods and Natural Language Processing, FSMNLP'09*, volume 6062 of *Lecture Notes in Artificial Intelligence*, pages 139–146. Springer Verlag, 2010. 2010.

[27] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA'06, pages 17–24. ACM, 2006.

[28] François Denis, Aurélien Lemay, and Alain Terlutte. Residual Finite State Automata. *Fundamenta Informaticae*, 51(4):339–368, 2002.

[29] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[30] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning Workflow Petri Nets. In *Proceedings of the* 31^{th} *Int. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency, Petri Nets'10*, Lecture Notes in Computer Science, pages 206–225. Springer Verlag, 2010. 2010.

[31] Lu Feng, Marta Z. Kwiatkowska, and David Parker. Automated Learning of Probabilistic Assumptions for Compositional Reasoning. In *Proceedings of the* 14^{th} *Int. Conf. on Fundamental Approaches to Software Engineering, FASE '11*, volume 6603 of *Lecture Notes in Computer Science*, pages 2–17. Springer Verlag, 2011.

[32] N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306(1-3):155–175, 2003.

[33] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[34] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the* 17^{th} *Int. Conf. on Tools and algorithms for the construction and analysis of systems, TACAS'11*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Verlag, 2011.

[35] Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *Proceedings of the* 20^{th} *Int. Conf. on Computer Aided Verification, CAV'08*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer Verlag, 2008.

[36] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *Procceedings of the* 31^{th} *Int. Conf. on Software Engineering, ICSE'09*, pages 430–440. IEEE Computer Society, 2009.

[37] E. Mark Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[38] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010.

[39] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of Event-Recording Automata Using Timed Decision Trees. In *Proceedings of the* 17^{th} *Int. Conf. on Concurrency Theory, CONCUR'06*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer Verlag, 2006.

[40] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Proceedings of the* 5^{th} *Int. Conf. on Fundamental Approaches to Software Engineering, FASE '02*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[41] Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001.

[42] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the* 10^{th} *European Software Engineering Conference held jointly with* 13^{th} *ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE'05*, pages 31–40. ACM, 2005.

[43] John E. Hopcroft and R. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. Technical Report 71-114, Cornell University, 1971.

[44] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[45] Falk Howar. Inferenz Parametrisierter Moore-Automaten. Master's thesis, TU Dortmund, Department of Computer Science, Chair of Programming systems, 2009.

[46] Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring Semantic Interfaces of Data Structures. *Accepted for ISoLA 2012*.

[47] Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. On Handling Data in Automata Learning - Considerations from the CONNECT Perspective. In *Proceedings of the* 4^{th} *Int. Symposium on Leveraging Applications, ISoLA'10 (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 221–235. Springer Verlag, 2010.

[48] Falk Howar, Maik Merten, Bernhard Steffen, and Tiziana Margaria. *Formal Methods for Industrial Critical Systems*, chapter Practical Aspects of Active Automata Learning. Wiley-VCH, 2012.

[49] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *Porceedings of the* 13^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI'12*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Verlag.

[50] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In *Proceedings of the* 4^{th} *Int. Symposium on Leveraging Applications, ISoLA'10 (1)*, number 6415 in Lecture Notes in Computer Science, pages 687–704. Springer Verlag, 2010.

[51] Falk Howar, Bernhard Steffen, and Maik Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *Porceedings of the* 12^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI'11*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer Verlag, 2011.

[52] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *Proceedings of 2003 International Test Conference, ITC'03*, pages 971–980. IEEE Computer Society, 2003.

[53] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-Specific Optimization in Automata Learning. In *Proceedings of the* 15^{th} *Int. Conf. on Computer Aided Verification, CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, 2003.

[54] Muhammad Naeem Irfan. State Machine Inference in Testing Context with Long Counterexamples. In *Proceedings of the* 3^{th} *Int. Conf. on Software Testing, Verification and Validation, ICST'10*, pages 508–511. IEEE Computer Society, 2010.

[55] Malte Isberner. Untersuchung der Optimierbarkeit regulärer Extrapolationsverfahren durch Ausnutzung vorhandenen Wissens. Master's thesis, TU Dortmund, Department of Computer Science, Chair of Programming systems, 2011.

[56] Malte Isberner, Bernhard Steffen, and Falk Howar. Inferring Automata with State-local Alphabet Abstractions. *Submitted to CAV 2012*.

[57] Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia

Bertolino, and Antonino Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *Proceedings of the* 14^{th} *IEEE Int. Conf. on Engineering of Complex Computer Systems, ICECCS'09*, pages 154–161. IEEE Computer Society, 2009.

[58] Bengt Jonsson. Learning of Automata Models Extended with Data. In *Formal Methods for Eternal Networked Software Systems, SFM'11*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer Verlag, 2011.

[59] M. Kaminski and N. Francez. Finite-Memory Automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[60] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.

[61] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Procceedings of the* 30^{th} *Int. Conf. on Software Engineering, ICSE'08*, pages 501–510. ACM, 2008.

[62] Oded Maler and Amir Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995.

[63] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *Proceedings of the* 9^{th} *Annual IEEE Int. High-Level Design Validation and Test Workshop, HLDVT'04*, pages 95–100. IEEE Computer Society, 2004.

[64] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation. In *Proceedings of 2005 Int. Test Conference, ITC'05*. IEEE Computer Society, 2005.

[65] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.

[66] Karl Meinke and Muddassar A. Sindhu. Incremental Learning-Based Testing for Reactive Systems. In *Tests and Proofs - 5th Int. Conf. TAP'11*, volume 6706 of *Lecture Notes in Computer Science*, pages 134–151. Springer Verlag, 2011.

[67] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *Proceedings of the* 17^{th} *Int. Conf. on Tools and algorithms for the construction and analysis of systems, TACAS'11*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer Verlag, 2011.

[68] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench User Manual (Version 7.1). Available from http://homepages.inf.ed.ac.uk/perdita/cwb/.

[69] A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[70] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, Germany, 2003.

[71] Object Management Group (OMG). CORBA. `http://www.corba.org/`. Version of 01.02.2012.

[72] D. Peled, M. Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[73] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.

[74] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.

[75] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[76] H. Sakamoto. Learning Simple Deterministic Finite-Memory Automata. In *Proceedings of the* 8^{th} *Int. Conf. on Algorithmic Learning Theory, ALT'97*, volume 1316 of *Lecture Notes in Computer Science*, pages 416–431. Springer Verlag, 1997.

[77] Muzammil Shahbaz and Roland Groz. Inferring Mealy Machines. In *Proceedings of the* 2^{th} *World Congress on Formal Methods, FM'09*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer Verlag, 2009.

[78] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and Integration of Parameterized Components Through Testing. In *Proceedings of the* 19^{th}*IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems,* 7^{th} *Int. Workshop, TestCom/FATES'07*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer Verlag, 2007.

[79] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning Parameterized State Machine Model for Integration Testing. In *Proceedings of the* 31^{th} *Annual Int. Computer Software and Applications Conference, COMPSAC'07*, pages 755–760. IEEE Computer Society, 2007.

[80] Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative refinement of specification for component based embedded systems. In *Proceedings of the* 20^{th} *Int. Symposium on Software Testing and Analysis, ISSTA'11*, pages 276–286. ACM, 2011.

[81] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the ACM/SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'07*, pages 174–184. ACM, 2007.

[82] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems, SFM'11*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Verlag, 2011.

[83] The Open Group. DCOM. http://www.opengroup.org/comsource/. Version of 01.02.2012.

[84] B.A. Trakhtenbrot and I.M. Barzdin. *Finite Automata; Behavior and Synthesis [By] B. A. Trakhtenbrot and Ya. M. Barzdin. Translated From the Russian by D. Louvish. English Translation Edited by E. Shamir and L. H. Landweber*, volume 1 of *Fundamental Studies in Computer Science*. 1973.

[85] W3C. Web services. http://www.w3.org/2002/ws/. Version of 01.02.2012.

[86] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting Object Usage Anomalies. In *Proceedings of the 6^{th} joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, ESEC-FSE'07*, pages 35–44. ACM, 2007.

[87] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT Int.łSymposium on Software testing and analysis, ISSTA'02*, pages 218–228. ACM, 2002.