Active automata learning for real-life applications

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

von
Maik Merten

Dortmund
2013

Tag der mündlichen Prüfung: 09.01.2013
Dekanin: Prof. Dr. Gabriele Kern-Isberner

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Reiner Hähnle

Acknowledgements

I want to thank Bernhard Steffen for guiding me during the past five years. Clearly, this dissertation is a result of having being challenged, motivated, and supported in a truly unique environment, created by the persons that gathered at the Chair of Programming Systems to create and cooperate.

Thus I would also like to thank my colleagues, and in particular Falk Howar, with whom I had the pleasure of sharing an office for three years. Hilarity sure ensued, as did pleasant traveling and countless hours of fruitful discussion.

Last but not least, thanks also go to my family, which was both supportive and patient.

Contents

1. Introduction 1
1.1. Research problems addressed in this thesis 3
1.2. Contributions 4
1.3. Organization 5

2. Preliminaries and basic concepts of automata learning 7
2.1. Mealy machines and running example 7
2.2. Passive automata learning 11
2.2.1. An algorithm for passive learning 11
2.2.2. Problems with passive automata learning 13
2.3. Active automata learning 14
2.3.1. The MAT model: introducing the teacher 14
2.3.2. The L^* algorithm 14

3. Tools for the real life: LearnLib and LearnLib Studio 19
3.1. LearnLib 19
3.1.1. LearnLib interfaces for MAT learning 20
3.1.2. LearnLib's extended MAT model 21
3.1.3. Query batches in LearnLib 22
3.1.4. Supporting infrastructure 23
3.1.5. The universality of LearnLib 23
3.2. LearnLib Studio 24
3.2.1. The LearnLib Studio modeling approach 24
3.2.2. SIB learning 27

4. The DHC algorithm 29
4.1. The core of the algorithm 29
4.2. Algorithm termination 30
4.3. Refining hypotheses 30
4.4. Memory consumption 33
4.5. Number of generated MQs 35
4.6. Creating query batches 36

5. Scalability in practice 37
5.1. Practical concerns 37
5.1.1. Challenges for active automata learning 37
5.1.2. Filter example: The query cache 39
5.2. Distributed learning 39

5.3. Moving learning into the cloud . . . 41
5.3.1. Cloud computing and automata learning . . . 41
5.3.2. The WebABC . . . 43

6. Automated configuration of learning setups **45**
6.1. The role of test drivers in active automata learning . . . 45
6.2. Learning setup creation by interface analysis . . . 46
6.2.1. Determining a learning alphabet . . . 47
6.2.2. Interfacing with the target system . . . 47
6.2.3. Dealing with live data values . . . 48
6.3. The learning setup interchange format by example . . . 49

7. Related Work **53**

8. Conclusion and Outlook **57**
8.1. Conclusion . . . 57
8.2. Outlook . . . 58

A. Selected papers **67**

B. Comments on my participation **69**

C. Other publications **71**

List of Figures

1.1. Software development on the UNIVAC computer system 2
1.2. Overview on presented contributions. 6

2.1. The illustrated state space of the coffee machine. Source: Paper I 8
2.2. Mealy specification of the coffee machine. Source: Paper I 9
2.3. Minimal Mealy machine of the coffee machine. Source: Paper I 11
2.4. Tree constructed according to six input/output traces, nodes colored after corresponding states (r)ed, (g)reen, (b)lue. 12
2.5. Automaton model constructed from the colored tree in Figure 2.4. 12
2.6. The structure of the L^* observation table. 15
2.7. A closed and consistent observation table. Source: Paper I 16

3.1. The MAT model components (solid border) and the respective interfaces of the LearnLib component model (dashed border). 20
3.2. Interplay of basic LearnLib components. 21
3.3. An overview of all basic LearnLib components. 23
3.4. Modeled learning setup with configuration phase and implicit query handling. 25
3.5. Modeled learning setup utilizing the new modeling paradigm. 26

4.1. First steps of DHC hypothesis construction. Source: Paper I 31
4.2. First hypothesis constructed by the DHC algorithm. 31
4.3. Pseudocode of the DHC core algorithm. Source: Paper III 32
4.4. An early stage of the DHC hypothesis construction, corresponding to Figure 4.1(c), with the distinguishing suffix "*water button*". Source: Paper I 33
4.5. Memory consumption of the DHC algorithm compared to two L^* implementations. Source: Paper III 35

5.1. Batch sizes for different algorithms implemented in LearnLib. Source: Paper IV 40
5.2. Scaling of different learning algorithms in a simulated distributed setup. . . 41
5.3. User interface of WebABC, with a modeled learning setup visible. 43

6.1. Architecture of a configurable test driver for active automata learning. Source: Paper V 46
6.2. Excerpt of a learning setup for the e-commerce example. Source: Paper V . 50
6.3. Learned model of the e-commerce example. Source: Paper V 52

1. Introduction

Without a doubt, it is fair to say that information technology made tremendous progress since its original inception. This is an easy to point out fact, given how our daily lives are obviously augmented with networked computer systems. Most of us find it nigh-impossible to escape the influence of the modern-day networked lifestyle, considering "offline time" to be mostly a part of the holiday season or vacations, offering a noticeable change in pace from everyday life.

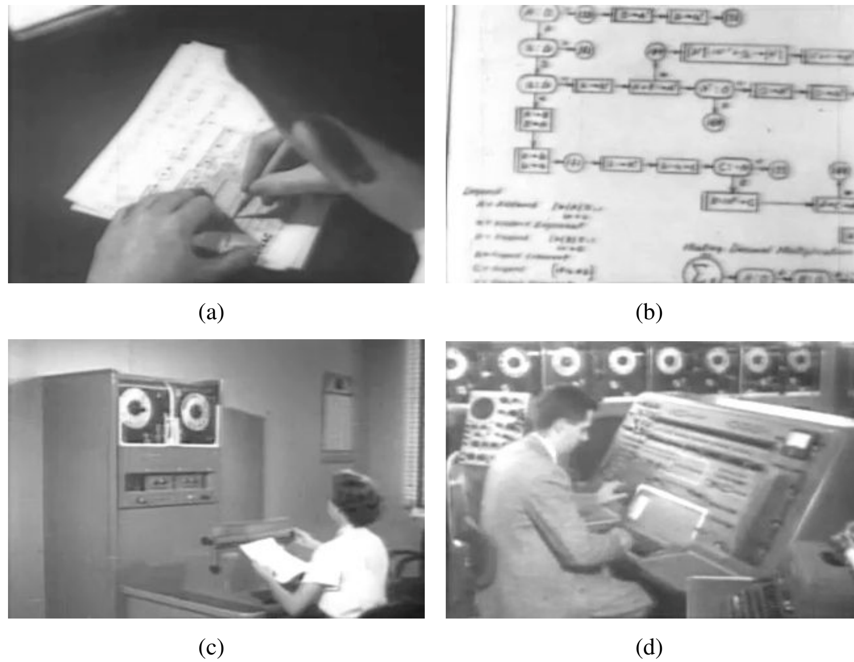
While considerable progress has been made in the construction of increasingly complex systems, mostly by introducing ever more layers of abstraction that are supposed to shield the engineers from the insurmountable flood of details of modern distributed and inherently complex systems, it appears that not every facet of software engineering is developing at the outstanding pace which seems to drive the industry.

As modern software development is mostly a "high level matter", using high level programming languages with rich libraries that facilitate several layers of abstraction, a single engineer will be hard-pressed to answer questions regarding the exact flow of computation and data processing. Be it object-relational database mappers, automatic memory management or "enterprise-enabled" management containers for business applications: the exact behavior in all but the most basic scenarios is usually not traceable by engineers employing these advanced technologies. This means that even isolated systems are not comprehensible in their entirety.

How much more difficult must it be to comprehend the implications of connecting several networked systems, all by themselves already only comprehensible on the surface? When focusing on what uncertainties linger beneath the surface, it must appear almost like a miracle that the very systems that play important roles in our daily lives evidently work - or at least work most of the time: incidents such as an email service provider employing "cloud storage" being forced to resort to tape backups to restore user accounts that were wiped by a software update (which happened to Google Mail in February 2011, [65]) are remarkably rare.

While this scenario of opaque networked systems may appear dire and intimidating, it is clear that the continued trend of moving system development onto a "higher level" by abstracting from the details of computer operation, despite having some undesired side effects, is an essential enabler for the continued evolution of software-engineering. It is hard to imagine how a flourishing software industry could have developed if engineers would have know every single detail of the hardware and basic operating system.

To illustrate this it can help to have a short glimpse on what software construction used to look like in the early days of automated electronic data processing. The UNIVAC system is regarded to be the first commercially available computer, introduced in 1951. Programs for this system were first drawn as algorithmic flow charts, which then were transformed into a sequence of instructions that subsequently were transcribed onto magnetic tape. The



(a) (b) (c) (d)

Figure 1.1.: Software development and operation on an UNIVAC system: (a) and (b) show the manual construction of program flowcharts. Transcription to magnetic tapes is shown in (c), software operation in (d). Screen captures taken from the advertisement film “Remington-Rand Presents the UNIVAC”.

instructions would encompass nearly all aspects of managing computing resources and effecting the actual computation. Debugging of programs was everything but an interactive process, involving lengthy roundtrips of design, transcription and operation. This development process is illustrated in Figure 1.1.

It nowadays appears impractical to create complex software systems in this fashion. However, as natural artifacts of the archaic development process useful side-products were created: the flowcharts of the programs can be considered to be at least a natural precursor of documentation, enabling other engineers to grasp the overall structure of the program. Of course, the algorithmic flowcharts already are an abstract view onto the actual computation, omitting details of machine operation.

Nowadays development of IT systems does not literally have to start at the drawing board. Nonetheless various modeling approaches, supposed to escort the development process from start to finish, have been proposed to ease design and documentation, with the intent of enabling creation of high quality software products in a methodical and transparent matter.

In practice it needs considerable discipline and effort to develop systems that way. Many systems are created by “hacking”, without an all-encompassing design effort. Programs that were merely intended to demonstrate the feasibility of a certain approach are developed into full-scale systems. Legacy systems with a history of neglected documentation are extended beyond their original design. Documentation is, if at all, usually an afterthought.

In fact, documentation duties are considered a nuisance amongst many programmers, which does not impair the ability to succumb to desperate outcries whenever they find documentation is missing for a system they are supposed to work on.

With system descriptions being usually incomplete or otherwise inadequate, methods to automatically infer formal models from preexisting systems in a "black-box" fashion, i.e., methods that can create formal descriptions even in the aftermath of system deployment, have immediate appeal. With active automata learning a method emerged in the 1980ies which shows great potential for creating formal models that can be used for documentation and testing. Nonetheless adaptation into real-life use has been slow, caused by considerable resource demands of the underlying algorithms as well as software implementations that were not focused on the exploration of real-life reactive systems. In this thesis refinements to algorithms and software implementations will be discussed that were created with the goal of real-life system exploration in mind.

1.1. Research problems addressed in this thesis

In this section the research problems addressed in this thesis are summarized. The central question of this thesis is

How can active automata learning be readied for application on real-life systems?

Automata learning is a concept discussed in the literature for decades. Accordingly, the theoretical framework for learning automata from observations has been in place already for a considerable time. Nonetheless considerable progress is still being made, e.g., by increasing the expressiveness of models that can be learned [33].

Despite the ever-increasing theoretical maturity of the field, real-life applications are few and far between. In part this can certainly be attributed to the lack of ready-made infrastructure, e.g., frameworks that support automata learning with the goal of learning realistic systems. Additionally, the degree of automation in this field is low, meaning that learning setups have to be instantiated manually and per-system, making this a time-consuming and laborious undertaking.

To make automate learning applicable to real-life scenarios, following tasks have to be solved:

Creating an infrastructure for real-life learning: For actual applicability of learning algorithms by engineers the mere existence of interesting algorithms in literature is not enough: clearly, the algorithms have to be implemented and readily available as well. This alone, however, is still not enough, as algorithms should be easily exchangeable within a given structure, to allow for quick experimentation and evaluation. This calls for a flexible framework that provides unified access to a selection of pre-implemented and well-tested learning algorithms.

This framework should abstract from the inner-workings of learning algorithms and provide an easy-to-use interface.

Apart from integrating learning algorithms, the framework should also provide support for components that accelerate learning by infusing application-specific knowledge into the learning process, e.g., so called *filters* have to find a natural place within learning setups. In active automata learning, where the learning algorithm can *query* the target system for additional information, the number of queries can quickly exceed several ten millions. This can dominate the required runtime of constructing a model representation of the target system, possibly clocking in at several hours or days, which is clearly undesirable. Filters can dramatically reduce the number of queries that have to be processed, reducing the required time for learning to conclude accordingly.

Other important components that should fit into the general framework are *equivalence tests*, which determine if the learned model is indeed a faithful representation of the target system. If a mismatch is detected learning will resume creating a more faithful model. Otherwise, the learning procedure is finished. Equivalence tests are central to the correctness of active automata algorithms and thus have great practical importance.

As the theoretical state of the art in automata learning progresses, e.g., by extending the expressiveness of models that can be learned, the learning framework should be able to allow implementation of such algorithms without major changes, meaning the framework should be engineered along concepts that are shared between many different concrete algorithms and serve as general underpinning of the employed learning method.

While learning frameworks do exist, they fail to offer all-encompassing solutions for real-life application, e.g., by only offering a set of distinct learning algorithms without presenting a unified view, including little useful infrastructure or by being cryptic and unintuitive to use.

Enabling automated learning experiments: Creating application-fit learning setups can be a time-consuming process, which includes analyzing the target system and determining a fitting set of configuration options for the actual learning process. This hampers the overall adoption of learning methods and is a complete deal-breaker in any scenario where arbitrary systems are to be learned in an automated fashion without manual intervention. Such a scenario is provided in the Connect project [39], where connectors between systems are to be synthesized from learned models automatically.

Enabling high-level integration into processes: The construction of models for preexisting systems can be part of greater schemes, where, e.g., the constructed models are part of regression tests or software synthesis (as, e.g., in the Connect project). Thus it is desirable to employ a service-oriented view onto learning as one part of the solution for a bigger task, where learning components can be orchestrated as part of an all-encompassing process.

1.2. Contributions

In this thesis my following contributions are presented:

Learning framework and tools: A new Java framework for learning algorithms was developed, structured after core components of the minimally adequate teacher (MAT)

paradigm for active automata learning. This framework lays the foundation for the implementation of both classical and advanced active learning algorithms. This work supplants an older version of LearnLib that was implemented in C++ and did not feature a strong component model.

Employing the reworked framework and accompanying algorithm implementations, LearnLib Studio, a service-oriented integrated environment for modeling and executing learning setups, was created on top of the jABC [63] framework.

Learning algorithms: A new active learning algorithm that does not employ a traditional observation table was developed and implemented for the new LearnLib framework. This algorithm is useful for both didactic and practical uses, being inspired by the structure of the well-known breadth-first search and clearly separating data structures with distinct purposes, making it easy to tailor configurations with different memory footprint by simplified externalization.

Scalability of learning solutions: The new LearnLib component model allows for easy integration of generic or application-specific filters that reduce the overall number of learning queries to be processed, which is a vital concern when learning real-life systems of considerable size. The framework integrates means for parallelized and networked execution of queries, leading the way towards learning setups that are executed within a cloud environment. Experiments have been conducted that show that formidable time savings can be achieved by parallelized execution. In addition, a prototypical Software as a Service (SaaS) application, named the WebABC, was developed that supports modeling and executing learning setups in web browsers utilizing scalable cloud computing infrastructure.

Automated configuration of learning setups: Writing test drivers and determining an application-fit learning alphabet can be laborious and thus time-consuming. A configurable test driver, containing means to translate between the abstract input- and output-alphabets of the learning algorithm and concrete system input and output values, was developed. By means of interface analysis an application-specific configuration for the reusable test driver can be created, making fully automated learning setups possible.

1.3. Organization

This thesis is structured as follows: In Chapter 2 the theoretical background and formalisms subsequently used will be discussed, introducing the automaton model predominantly used in this thesis, followed by the foundations of automata learning. Chapter 3 discusses the structure of LearnLib, the automata learning library which served as implementation framework for this thesis. This section also introduces LearnLib Studio, a graphical tool for modeling and executing learning setups, built on top of LearnLib. A new learning algorithm first introduced in LearnLib will be presented in Chapter 4. After sketching challenges of automata learning in real-life practice and possible solutions in Chapter 5, the automation of learning setups is discussed in Chapter 6. Related work is discussed in Chapter 7. This thesis concludes in Chapter 8.

The structure of this thesis reflects the structure of the contributions.

Innovations to the structure of the underlying LearnLib framework and surround tools are the foundation for the following contributions. Subsequent innovation in the algorithm space was realized on top of this reworked learning infrastructure. With the renovated framework and associated algorithm implementations, attacking scalability issues of automata learning in general becomes the main focus. Lastly, with a learning stack enabled for real-life applications, the overall effort for producing and executing learning setups can be decreased substantially by automated configuration of learning setups, which can potentially drive the adoption of automata learning in practice.

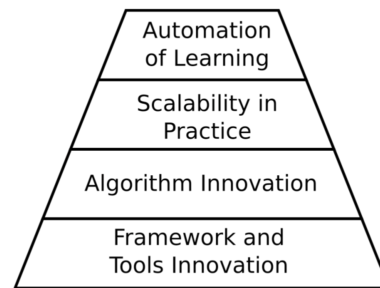


Figure 1.2.: Overview on presented contributions.

A graphical representation of this structure is provided in Figure 1.2. Variations of this figure will indicate the location of the respective chapters within this landscape.

Note on references: Subsequent sections will frequently refer to a selection of publications, denoted as “Paper I” to “Paper VI”. These references are declared in Appendix A.

2. Preliminaries and basic concepts of automata learning

In this chapter, basic concepts will be discussed that establish a background for the contributions discussed in this thesis. This includes a description of the machine model mostly discussed throughout this work, i.e., Mealy machines, and the overall concept of automata learning, both in passive and active form.

2.1. Mealy machines and running example

Having formal representations of systems is a prerequisite for verification techniques such as model checking. In this thesis, *system* usually refers to *reactive systems*. A very simple example system, taken from Paper I, is the following:

Example 1 (A coffee machine) *Let us consider a very simple reactive system: a coffee machine. This machine has an assessable user interface, namely a button which starts the production of delicious coffee. However, before the production of this precious fluid can commence, a water tank (filled with water) and a coffee pod have to be put in place. After every cup of coffee produced, the machine has to be cleaned, which involves the removal of all expendables. Thus the operations possible on the machine are "water" (fill the water tank), "pod" (provide a fresh coffee pod), "clean" (remove all expendables) and "button" (start the production of coffee).*

One single flaw that escaped product testing, however, is that the machine will immediately enter an error state on any mishandling. If, e.g., the button for coffee production is pressed before a complete set of expendables is filled in, an error will be signaled that cannot be overcome using the conventional interaction operations described above. This explains the lukewarm reception by consumers and in turn the affordable price of the machine.

The state space of the machine is readily observable (see Figure 2.1), as is the output produced: the machine can be "OK" ("✓") with a user interaction, produce coffee ("☕"), or express its dissatisfaction with the way it is operated by signaling an error ("✷"). □

A very widespread automata model are Deterministic Finite Automata (DFAs), which model languages, i.e., decide if a sequence of input symbols is recognized as being included in a target language or not. DFAs are well-studied objects in the field of language theory that recognize *regular languages*. Algorithms exist, e.g., for automata products and for minimization, leading to canonical forms.

In practice, when trying to employ DFAs to model reactive systems, the limitation of DFAs to merely *accept* or *reject* sequences of input symbols poses certain challenges: Most reactive systems have no natural notion of accepting or rejecting inputs and instead produce

(a) empty (b) with pod (c) with water

(d) with pod and water (e) success (f) error

Figure 2.1.: The illustrated state space of the coffee machine. Source: Paper I

reactions from a – sometimes large – domain of possible values, i.e., an *output alphabet*. For instance, the coffee machine example already possesses three observable outputs. One way to cope with this problem is to explicitly encode all expected pairs of input- and output-symbols into the alphabet Σ and to denote the production of an input/output-pair by transitioning into an accepting state.

This construction, obviously, is tedious and verbose. To make matters worse, the resulting model does not exactly lend itself to easy comprehension, due to being inflated by the very construction principle. Clearly, a more fitting model for reactive systems is desirable.

It turns out that *Mealy machines* are a much better fit for reactive systems:

Definition 1 *A Mealy machine is a tuple* $\langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ *with:*

- S*, a finite set of states, also called locations,*
- $s_0 \in S$*, the initial state,*
- Σ*, a finite set of input symbols, the input alphabet,*
- Ω*, a finite set of output symbols, the output alphabet,*
- $\delta : S \times \Sigma \to S$*, a function specifying the transitions for every state to its successor states, and*
- $\lambda : S \times \Sigma \to \Omega$*, a function producing an output symbol for every transition.*

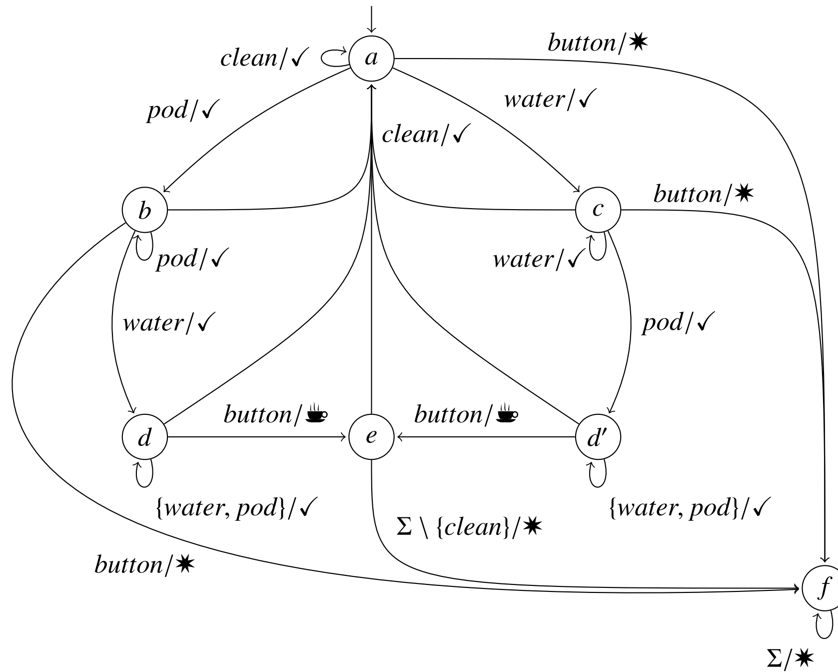


Figure 2.2.: Mealy specification of the coffee machine. Source: Paper I

Intuitively, when providing an input symbol $\alpha \in \Sigma$, a Mealy machine will proceed into a successor state as denoted by the δ function and produce an output symbol $o \in \Omega$ while doing the transition, as denoted by λ. Both δ and λ can be extended in an intuitive way to also process non-empty sequences (words) of input symbols $w \in \Sigma^+$ by successive application to every symbol in the input word. In this thesis both versions of these functions will be used, depending on context.

It is apparent that Mealy machines are a conservative extension of DFAs, with the key differences being:

- Instead of being restricted to $\{+, -\}$ as output alphabet, arbitrary finite sets are possible with Mealy machines.
- The "output" of the machine is a property of the transition taken, not of the state entered.

The freedom of defining application-fit output symbols significantly eases modeling real-life systems. One notable property that does not change when going from DFAs to Mealy machines, however, is the class of languages that can be expressed: every DFA can be transformed into a Mealy representation (this is trivial) and every Mealy machine can be represented by a DFA (by means of the cumbersome construction outlined above, where the input alphabet consists of pairs of input and output symbols).

Example 2 (The coffee machine Mealy model) *The behavior of the coffee machine described in Example 1 can be specified as Mealy machine* $\mathcal{M}_{cm} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$*, with*

- $S = \{a, b, c, d, d', e, f\}$
- $s_0 = a$
- $\Sigma = \{water, pod, button, clean\}$
- $\Omega = \{\checkmark, ☕, ✷\}$

The model presented in Figure 2.2 defines the output- and transition-function. □

For algorithms constructing automata models, the definition of equivalence between states is of great importance. For Mealy machines, equivalence between states is defined as follows:

Definition 2 *Two states $s_a, s_b \in S$ of a Mealy machine are* equivalent *if both states show exactly the same behavior for all* futures*, i.e., every input word $w \in \Sigma^+$ produces the same output for both:*
$s_a \equiv s_b \iff \lambda(s_a, w) = \lambda(s_b, w)\ \forall w \in \Sigma^+$

In automata models, states can be identified by the words that reach them, i.e., by their *access sequences*. Thus this notion of state equivalence can be related directly to the Nerode relation on languages:

Definition 3 *Two words $x, y \in \Sigma^*$ are considered equivalent by the Nerode relation on the language L if for any suffix $z \in \Sigma^*$ the concatenations xz and yz are either members or non-members of language L:*
$x \sim_L y \iff (\forall z \in \Sigma^* : xz \in L \iff yz \in L)$

With this relation, equivalence classes on words can be defined:

Definition 4 *For $x \in \Sigma^*$, the equivalence class $[x]$ is defined as*

$[x] = \{y \in \Sigma^* \mid x \sim_L y\}$

Applying the Nerode equivalence classes, all states whose access sequences are in the same equivalence class are considered equivalent. Thus the number of states in the minimal automaton representation for a given language L equals the number of Nerode equivalence classes. In fact, a language is only regular if a finite state acceptor can be constructed, i.e., the number of equivalence classes is finite (Myhill-Nerode theorem, cf. [52]).

Referring again to the coffee machine example illustrated in Figure 2.2, it can be observed that the states d and d' produce the same output behavior for all futures. This means that these two states are equivalent according to the specification of state equivalence for Mealy machines. Thus, the states d and d' can be merged, resulting in a minimal Mealy machine representation displayed in Figure 2.3.

The concept of (non-)equivalence of states is a very fundamental principle for automata learning. By identifying states by means of prefixes (i.e., their access sequences) and proving inequality by diverging output on suffixes (their futures), the state space of the model under construction can be built and successively refined as new data is gathered.

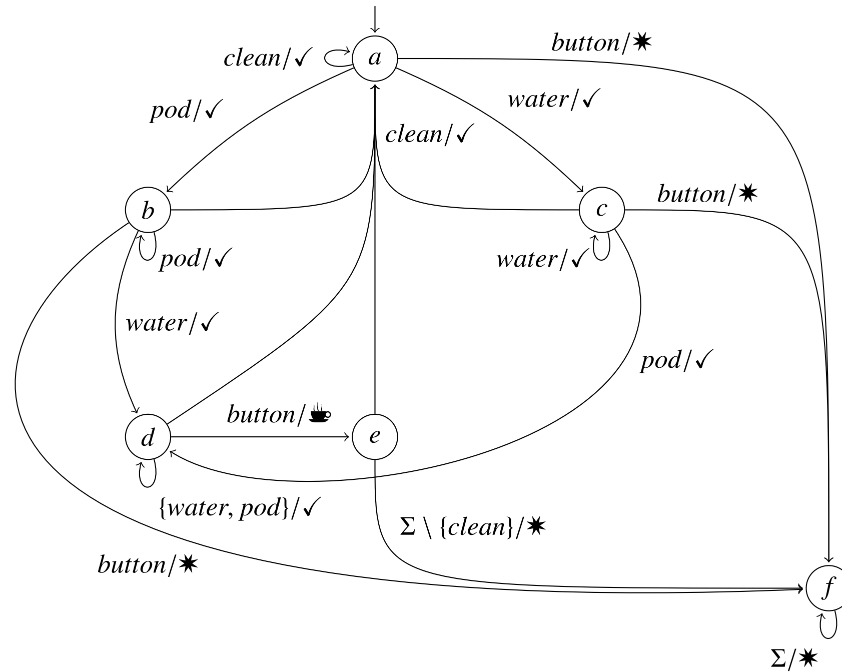


Figure 2.3.: Minimal Mealy machine of the coffee machine. Source: Paper I

2.2. Passive automata learning

In automata learning two distinct, but complementary, branches have emerged, each possessing unique advantages and disadvantages: passive learning and active learning.

Passive automata learning constructs automata representations of prerecorded behavioral traces of systems. Each trace is composed of input symbols, denoting stimuli the system was exposed to, and corresponding output symbols denoting the system's reaction to the input symbols. One example for a typical source of such traces are system logs which were created during normal operation of a system.

2.2.1. An algorithm for passive learning

One simple way to create deterministic automata models from prerecorded traces is illustrated in Figure 2.4: the traces are used to construct a data structure resembling a prefix tree. Nodes in this tree are associated with states in the automaton model to be constructed, with edges between nodes denoting state transitions for a given input symbol, producing one output symbol in the process. The mapping between nodes and associated states is determined by solving a graph coloring problem, where following rules are applied to generate deterministic automata models:

- All nodes associated with the same color produce the same output symbol for any given input symbol.
- For any given input symbol, the transition function for all equally colored nodes targets successors with the same color.

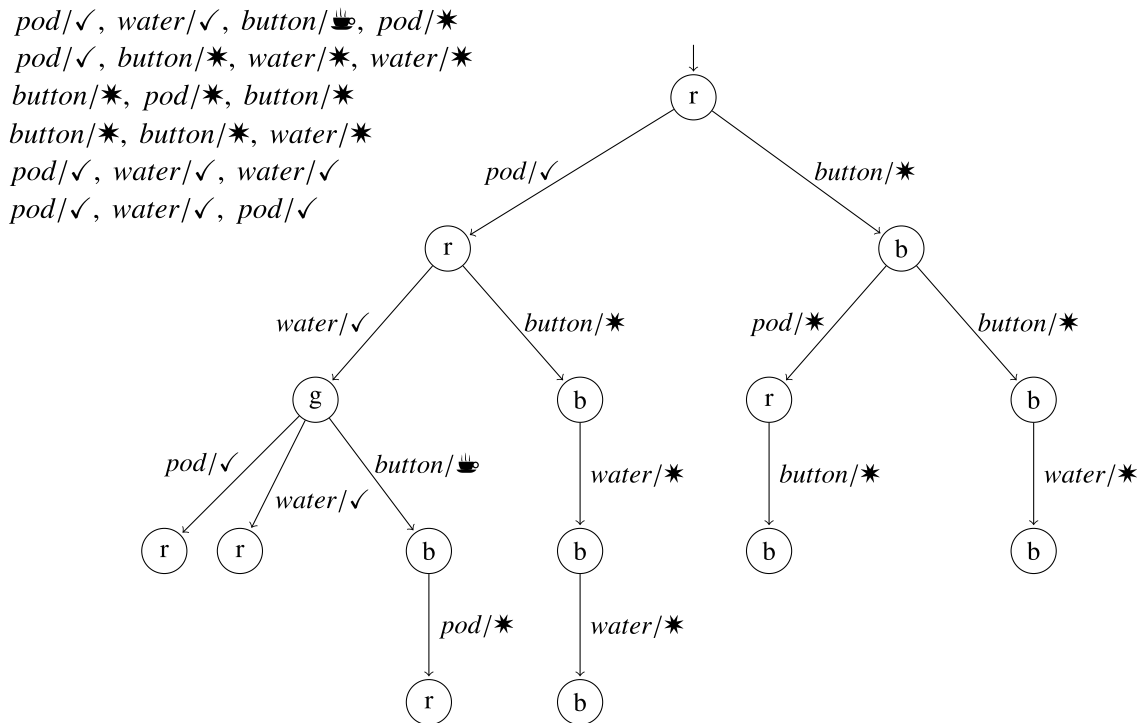


Figure 2.4.: Tree constructed according to six input/output traces, nodes colored after corresponding states (r)ed, (g)reen, (b)lue.

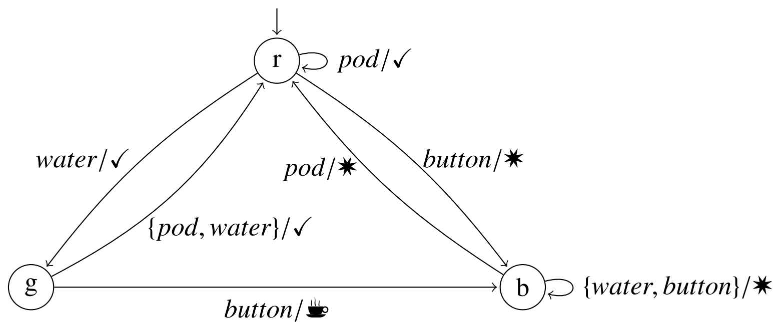


Figure 2.5.: Automaton model constructed from the colored tree in Figure 2.4.

Figure 2.5 shows the final result of the automaton model construction process for the traces given in Figure 2.4. As can easily be seen, the model can successfully reproduce the traces that were provided as construction input and can also serve as a predictor for as-of-yet unseen behavior. Seen from this perspective, the presented approach appears to be a complete and resounding success. However, this construction method is not without its fair share of problems, as will be discussed in the following.

2.2.2. Problems with passive automata learning

One issue with the presented approach is the runtime complexity. It is well-known that the graph-coloring problem is NP-complete. Thus, constructing a minimal solution to this problem can have undesirable runtime properties. In fact, it has been shown that constructing minimal automata from provided data is in general NP-complete [26] and not merely an artifact of the concrete approach chosen here. This high complexity means that two conflicting interests arise: the more data (in the form of system traces) is provided, the better are the chances that enough information is contained to construct an accurate model. However, this also makes finding an accurate solution increasingly difficult.

Another issue is that the produced result is ambiguous. In the graph coloring phase of model construction several alternatives exist that are not in contradiction to the construction rules. For instance, in Figure 2.4 the node that is reached by the input trace "*button*, *pod*" is colored red. This decision, however, is arbitrary, as, e.g., that particular node could just as well be colored blue. This has direct impact on the constructed automata model: the predictions the model produces can change significantly, depending on arbitrary choices during the construction process.

This issue makes it hard to judge the actual quality of the produced model as a predictor for future, i.e., of yet unobserved, behavior of the targeted system. Only once additional traces are collected from the target system and subsequently compared to model predictions the quality of the constructed model can be assessed.

It is important to note that this problem does not stem from a sloppy construction approach, but is instead caused by gaps in system knowledge contained in the traces employed as construction input: there is not enough information to construct an unambiguous solution. Referring to the example above, both decisions (either coloring the mentioned node red or blue) are completely valid given the provided information.

With additional system-specific information it may be possible to introduce additional construction rules aimed at procuring a result that is more likely to resemble the actual model representation of the target system, but in the end any such approaches still remain heuristic in nature. As such, it remains difficult to make informed statements on the accuracy of the produced automata model.

To produce models with ascertainable properties, it has to be assured that enough information on the target system is provided so that no ambiguity remains in the construction process. This can be achieved by producing "characteristic" traces that leave no room for uncertainty [20]. In practice, however, when learning actual real-life systems, it is hard to produce traces that indeed are characteristic.

Another idea is to request additional information wherever ambiguity exists. This is the approach chosen by active automata learning.

2.3. Active automata learning

Active automata learning is often associated with Dana Angluin's seminal algorithm L^* [6]. Angluin introduced the concept of a minimally adequate teacher (MAT), which provides information about the System Under Learning (SUL) so that accurate construction of models can be guaranteed.

2.3.1. The MAT model: introducing the teacher

In the MAT model of automata learning, the teacher can answer two distinct sorts of queries. As the focus of this work is on learning automata models for reactive systems, all formalisms will be provided according to the definition of Mealy machines which was given in Section 2.

- *Membership Queries (MQs)*: Provided a sequence of input symbols, the teacher will return system output accordingly. This means that for an input word $w \in \Sigma^+$ the teacher will produce $MQ(w) \in \Omega^+$.
- *Equivalence Queries (EQs)*: Provided a learned Mealy machine hypothesis $M^H = \langle S^H, s_0^H, \Sigma, \Omega, \delta^H, \lambda^H \rangle$, the teacher will produce a *counterexample* $c \in \Sigma^+$ that produces output on M^h that is different from the output produced on the correct Mealy machine representation of the target system $M^S = \langle S^S, s_0^S, \Sigma, \Omega, \delta^S, \lambda^S \rangle$, i.e., $\lambda^H(c) \neq \lambda^S(c)$. If M^H and M^S are equivalent, meaning that the learned hypothesis is an accurate representation of the target system and thus no such $c \in \Sigma^+$ exists, no counterexample will be produced.

Using these two types of queries, algorithms in the MAT model can be separated into two distinct phases:

1. construction of a hypothesis automaton using MQs
2. validation of the hypothesis using EQs

These phases are executed in turn. If a counterexample was produced via EQ in the second phase, the first phase will be resumed, constructing a refined hypothesis model. If no counterexample is produced, the learning procedure terminates. Given that EQs will only produce no counterexample if the learned hypothesis model is correct, it is easy to see how this procedure is trivially *partially correct*, as the procedure will only terminate with a correct result.

In the following, the L^* algorithm will be sketched as one example of algorithms following the MAT model. An in-detail construction analysis of L^* is provided in Paper I.

2.3.2. The L^* algorithm

As described above, the states of an automaton can be characterized by sets of words in the following fashion:

1. By prefixes that reach a specific state, i.e., its access sequences, and

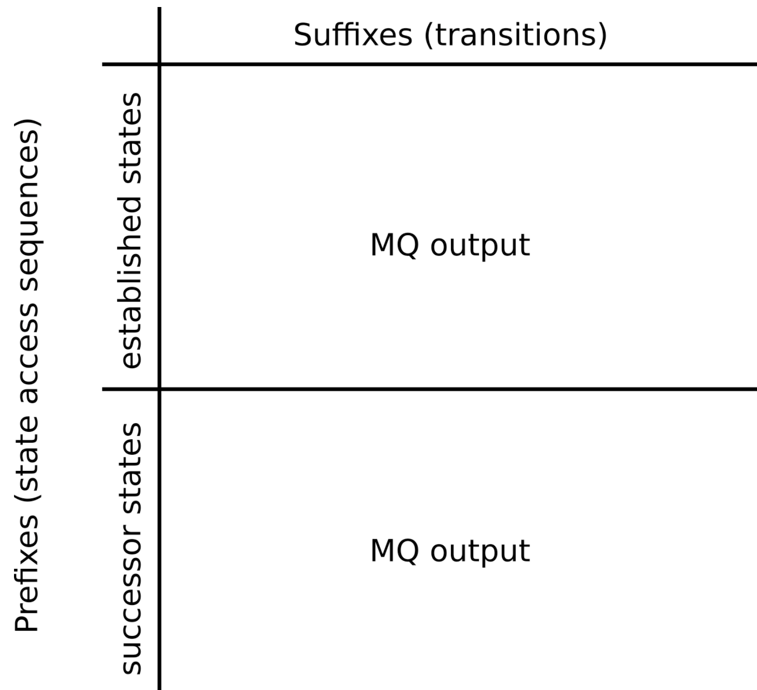


Figure 2.6.: The structure of the L^* observation table.

2. by an additional set of suffixes that denote state transitions.

Angluin's L^* algorithm organizes these pre- and suffixes in a data structure called *observation table*. In this table, which is illustrated in Figure 2.6, prefixes (that in effect are access sequences to states) are organized in rows, while the columns are dedicated to suffixes (and thus state transitions). The cells in the table represent output behavior for a given state and a given transition and are filled by means of MQs, with the query word being the concatenation of the pre- and suffix for the given cell.

Rows with the same cell content, i.e., states that show the same output behavior for all suffixes, are considered equivalent by the L^* algorithm.

The observation table is split in two halves: the upper half contains all rows associated with access sequences belonging to established states; the lower half contains all rows belonging to the transition successors of the states of the upper half. Any new states discovered in the lower half (i.e., a row with content that is not already present in the upper half) will be moved to the upper half.

Algorithm termination

L^* will terminate once the observation table is *closed* and *consistent*. These two properties are defined as follows:

- An observation table is closed if all transitions lead to already established states. Closedness is established by moving all new discovered states into the upper half of the observation table and exploring their successors, filling the observation table accordingly.

- An observation table is consistent if all rows (i.e., states) with the same row content (i.e., states that are considered equivalent) have equivalent successors for every transition. If an observation table is not consistent, this means that at least one state has two distinct successors for one transition, i.e., the automaton shows nondeterminism. Inconsistency is resolved by adding suffixes witnessing inequality of offending states.

If the observation table is closed and consistent, a deterministic conjecture can be constructed. An example is provided in Figure 2.7, showing the observation table corresponding to the first hypothesis produced when learning the coffee machine. It can be observed that only two states were identified, while the target system possesses six states, demonstrating the need of a refinement mechanism.

		$\mathcal{D}$			
		water	*pod*	*button*	*clean*
$\mathcal{S}p$	ϵ	✓	✓	✷	✓
	button	✷	✷	✷	✷
$\mathcal{L}p$	*water*	✓	✓	✷	✓
	pod	✓	✓	✷	✓
	clean	✓	✓	✷	✓
	button · water	✷	✷	✷	✷
	button · pod	✷	✷	✷	✷
	button · button	✷	✷	✷	✷
	button · clean	✷	✷	✷	✷

Figure 2.7.: A closed and consistent observation table. Source: Paper I

Refining hypotheses

Every conjecture will be subjected to verification by means of an EQ. If a counterexample is found, Angluin's L^* will add rows for all prefixes of the counterexample. As the counterexample will reach at least one state which was wrongfully identified as equivalent to another state (but is evidently not, as witnessed by the diverging output), this means that the table will now be inconsistent. This inconsistency will be resolved by introducing new suffixes as described above.

In short, refinement of the hypothesis will be achieved by introducing new rows and – in turn – columns to the observation table. To achieve closedness of the observation table, new MQs will be generated, filling the new table cells.

Memory consumption

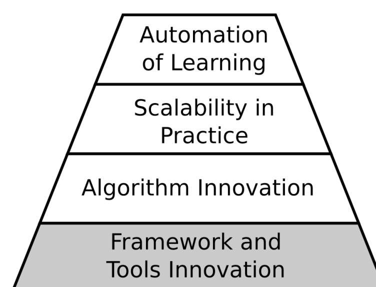
The observation table will store the results of every MQ generated by the learning algorithm. It can be proved that the maximum number of MQs generated by a learning setup utilizing an optimized version of the L^* algorithm with improved counterexample handling is $O(n^2k+k^2n+n\log(m))$, where n is the number of states in the automaton representation of the SUL, k is the size of the alphabet and m is the length of the longest counterexample discovered by an EQ (cf. Paper I).

The $n \log(m)$ term is due to an advanced strategy of counterexample evaluation and does not contribute to the memory consumption of the L^* core algorithm. Nonetheless, being quadratic in memory consumption regarding the size of the target system and its alphabet, the L^* algorithm can quickly consume undesirable amounts of system memory. This can severely restrict the maximum size of systems that can be explored by this algorithm as originally proposed by Angluin. Refinements to the basic algorithms, however, can in practice dramatically reduce the memory consumption encountered in real-life examples, e.g., by organizing the observation table data structure so that only one table row is allocated per discovered state, avoiding duplicate row contents.

3. Tools for the real life: LearnLib and LearnLib Studio

Although, as mentioned in the introduction, both the theoretical foundations of automata learning and learning algorithms based on the MAT paradigm have been known for a considerable amount of time (the former being based on language theory and especially the Nerode relation described in 1958 [52], the latter having been introduced by Angluin in 1987 [6]), the application on real-life systems is comparatively recent.

Likely reasons include the demands of real-life learning scenarios on computation and memory resources, only becoming satisfiable decades after the initial conception of the underlying concepts. Other factors may include the lack of easily available and robust algorithm implementations and supporting infrastructure.



This chapter describes the LearnLib, a library of learning algorithms and supporting infrastructure, and LearnLib Studio, a modeling and execution environment for learning setups. In the overall scheme of things, these components lay the foundation for other contributions presented in this thesis.

3.1. LearnLib

To make automata learning useful in practice, robust and versatile implementations of learning algorithms are obviously needed. However, algorithms alone are not sufficient, as surrounding infrastructure is needed as well to allow for quick assembly of learning setups. As catering both aspects represents a significant investment, having a flexible and thus reusable library, encompassing as many aspects of active automata learning as possible, has great value.

To satisfy the need for a flexible and powerful framework for automata learning, the LearnLib library was developed at the Chair of Programming Systems at TU Dortmund. Originally having been implemented in C++, the library was subjected to a large scale reengineering effort by the author of this work. Major goals for this reengineered version were ease of real-life deployment and modularization of the library to a much greater extend compared to the preexisting iteration. To achieve these goals, common patterns for all active automata algorithms had to be identified and transferred into a modular design. The result is a Java class library achieving high flexibility by condensing learning concerns into interfaces that allow for easily interchangeable implementations of learning algorithms and supporting infrastructure.

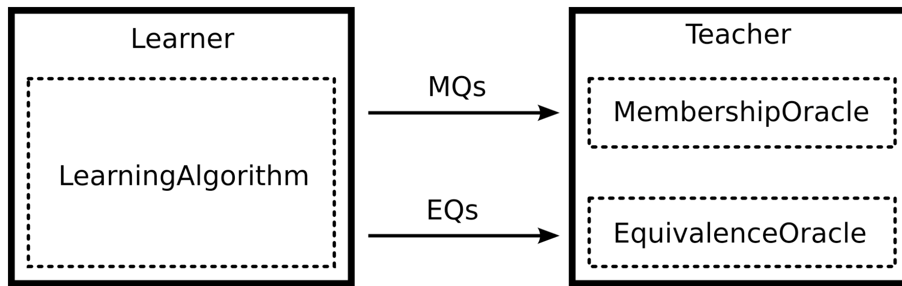


Figure 3.1.: The MAT model components (solid border) and the respective interfaces of the LearnLib component model (dashed border).

When analyzing active automata learning setups it is easy to realize recurring patterns that are shared regardless of the intended scope of application. In the MAT approach, learning setups have to provide solutions for the following tasks:

- Constructing queries and conjectures, a task which represents the main operation of any learning setup.
- Determining output for queries, i.e., active interrogation of the SUL, which necessitates means of interacting with the target system.
- Finding counterexamples to disprove inadequate hypotheses. This, again, requires interaction with the target system in most cases.
- Processing counterexamples, i.e., extraction of information from counterexamples that enables successive refinement of inadequate hypotheses.

These tasks have to be supported by an infrastructure that offers, e.g., means to assemble symbols from the learner's alphabet to queries and to create formal hypotheses. In LearnLib, components accounting for these basic tasks are standardized via interface contracts, which will be illustrated in the following.

3.1.1. LearnLib interfaces for MAT learning

The components described in the MAT model can be translated into interfaces encapsulating the respective functionalities in a straightforward way:

- `LearningAlgorithm`: This interface encapsulates learning algorithms such as L^* that pose queries and construct conjectures. The interface specifies methods to start the construction of queries (and therefore conjectures), to access hypotheses, and to submit information extracted from counterexamples to the learning algorithm.
- `MembershipOracle`: Components that provide answers for learning queries are described by this interface. Given the clearly defined scope, the interface is comprised of a single method that receives queries and returns a corresponding response. Test drivers instrumenting the SUL implement this interface, as do filters that answer queries with the help of preexisting or accumulated knowledge. An example for the latter are query caches that try to answer queries from the responses of preceding queries.

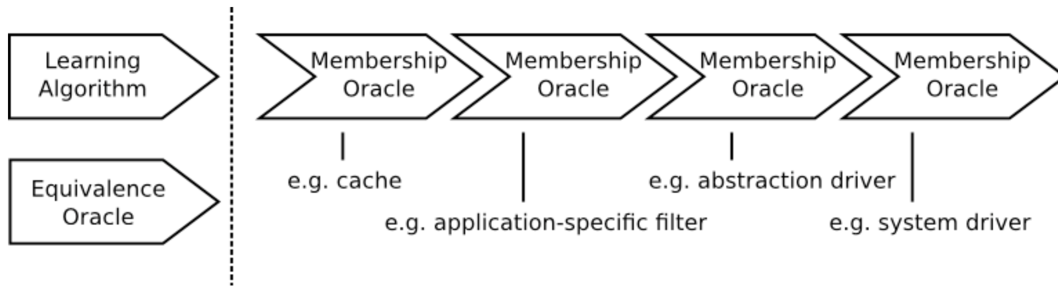


Figure 3.2.: Interplay of basic LearnLib components.

- `EquivalenceOracle`: Implementations of this interface receive a learned hypothesis and return a counterexample when a mismatch between the learned model and the SUL is detected. Thus these components implement EQs, usually employing approximative methods.

In Figure 3.1 the relation between the MAT model and the mentioned interfaces is illustrated, while Figure 3.2 shows how these basic components can be combined to create complete learning setups: a `LearningAlgorithm` component can be attached to a chain of `MembershipOracle` components, implementing, e.g., general or application-specific query filters or the actual SUL instrumentation. Every component of this chain tries to determine an answer for the current query, only delegating the query to the next `MembershipOracle` component if no answer could be constructed. This way the SUL adapter will only need to query the actual target system if all preceding filters failed to generate a response, potentially resulting in considerable time savings if invocation of the target system is time consuming.

As indicated in Figure 3.2, this "pipeline" for processing queries can also be used for EQs, which often are approximated by means of MQs.

3.1.2. LearnLib's extended MAT model

When implementing modern learning setups, one can identify an entity that is distinct from the learner and the teacher that already are present in the classical MAT model. This component is concerned with the analysis of counterexamples to ensure their efficient exploitation. In this area considerable progress has been made over the original conception of the L^* algorithm.

Departing from the L^* approach of adding prefixes of counterexamples to the observation table, subsequent algorithm proposals construct refined hypotheses following the addition of suffixes of the counterexample to the observation data structure, serving as witnesses for state non-equivalence. A simple way to ensure that all relevant suffixes are added is to simply add all suffixes of the counterexample. A less wasteful way of handling counterexamples is to determine which suffixes are essential for hypothesis refinement (and thus learning progress) and to only add those to the learner's data structures.

This can be achieved, e.g., by a binary search over the provided counterexample, which generates additional MQs to pinpoint the exact spot where the hypothesis and the SUL are in disagreement [60]. With this analysis a single suffix is identified that guarantees learning progress.

The analysis of counterexamples is encapsulated by the `CounterExampleHandler` interface in LearnLib. Components implementing this interface can be utilized by different learning algorithms (for instance, the procedure sketched above was originally proposed as an extension to the L^* algorithm, but can also be applied to other algorithms), supporting the idea that entities concerned with counterexample analysis are distinct from learners.

It is possible to regard counterexample analysis as a task that can be delegated to the teacher, in addition to answering MQs and EQs. This, however, appears unnatural as an "analysis query" on counterexamples differs significantly from MQs and EQs already handled by the learner: while the latter two provide information on the SUL, with results being predetermined by its behavior, the analysis of the counterexamples can yield significantly different results, depending much more on the strategy employed than on the SUL's behavior.

Another reason for not attributing counterexample analysis to the learner is that the analysis methods that have been conceived do not need extended teachers: they gather information about the SUL needed for analysis by means of standard MQs. Thus, looking from a practical standpoint, it appears most natural to regard counterexample analysis as a new component in the MAT model.

In real-life learning scenarios, where EQs can often be only approximated by additional MQs, this has interesting implications on the overall information flow: while in the MAT model as originally conceived there is only one active component (the learner) that queries the teacher using two types of queries, in practice there emerge two components (the learner and the counterexample analysis component) that query the teacher only by means of MQs.

3.1.3. Query batches in LearnLib

LearnLib supports a special delivery form of learning queries, the so named *query batches*. Batches of queries are collections of several distinct queries, which are emitted at once by a learning algorithm. It can be advantageous to process batches of learning queries instead of single queries, e.g., by sorting batches so prefixes of other queries within a batch are queried last. This can improve the hit rate of query caches and thus decrease the accumulated total time spent on retrieving query answers.

Another application of query batches is the distribution of queries onto several duplicated instances of the SUL. In this scenario, which will be discussed in more detail in Section 5.2, a networked cluster of target systems is utilized to retrieve query answers in parallel. Depending on the number of available target systems and the size of generated query batches, this can drastically decrease the wall clock time consumed for constructing conjectures.

Several learning algorithms in LearnLib support the assembly of learning queries into batches, filling, e.g., several cells in an observation table at once.

To process batches, an extended version of the `MembershipOracle` interface exists, which is named `MultiMembershipOracle`. Oracles implementing this interface can process a set of query words and will return a map data structure where each query is mapped to its respective output.

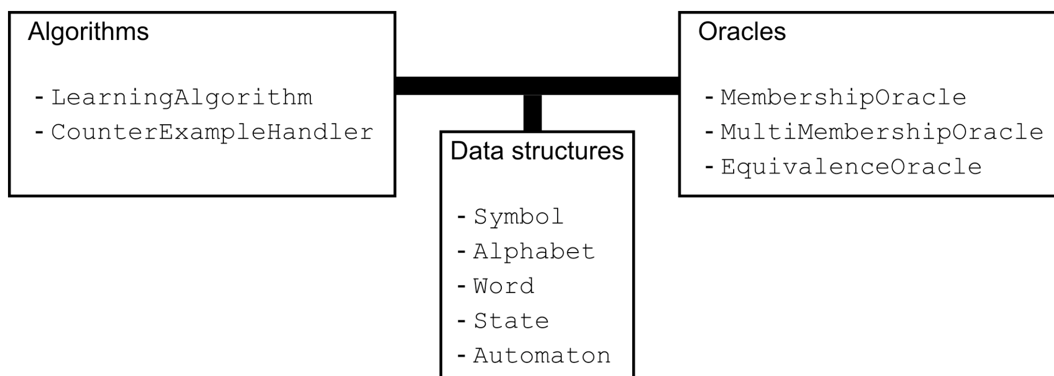


Figure 3.3.: An overview of all basic LearnLib components.

3.1.4. Supporting infrastructure

Augmenting the interfaces that devise contracts for algorithmic components, basic data structures upon which the algorithms can work on have to be provided as well. Of those, the following ones are the most notable:

- `Symbol`: Implementations of this interface can be used to, e.g., represent the symbols of the (often abstract) alphabet the learning algorithm uses for queries as well as the (concrete) symbols executable on the SUL.

- `Alphabet`: An alphabet is a set of `Symbol` objects. Alphabets are used to define, e.g., the input alphabet Σ of Mealy machines and thus form the basis for model construction.

- `Word`: Words are ordered sequences of `Symbol` instances. Queries generated by learning algorithms are instances of `Word`, as are the corresponding responses created by `MembershipOracle` components.

- `State` and `Automaton`: To allow for the construction of automaton conjectures, LearnLib includes a simple automaton library. The `Automaton` interface specifies a contract for automaton structures that `LearningAlgorithm` instances can use to form and provide hypotheses. For example, the `Automaton` interface defines that automaton states are to be represented by `State` objects, which utilize `Symbol` objects to define output and transition functions.

LearnLib includes implementations of these basic interfaces to facilitate learning of reactive systems, with Mealy machines as targeted formal model. However, it is possible to implement learning algorithms for other automaton models as well, provided implementations of the basic interfaces, engineered to cater the needs of the respective automaton model, are supplied.

3.1.5. The universality of LearnLib

As described in the previous sections, LearnLib is structured according to the principles of the MAT model. The interfaces have been designed to isolate reusable components, which

can be employed in a wide range of learning setups. As such the framework easily supports learning algorithms such as L^* or Observation Packs [32], with, e.g., counterexample analysis strategies being exchangeable between algorithms. An overview of the component concept is provided in Figure 3.3.

The first learning algorithms implemented within the new LearnLib component model were geared towards learning Mealy machine models of reactive systems. Testimony to the overall flexibility of the LearnLib framework is the arrival of algorithms that learn models as Register Automata (RAs, [17]), which extend the expressiveness of models over Mealy machines. The learning algorithm for RAs is formulated as MAT learner, meaning that the very same principles apply. Thus it was possible to fit an implementation of this algorithm into the framework merely by creating extended implementations of the supporting model infrastructure, such as `Automaton` or `Symbol`, to accommodate the extended machine model of RAs. The introduction of Register Automata to LearnLib was a subject of Paper II.

3.2. LearnLib Studio

LearnLib is augmented by LearnLib Studio, which allows for graphical composition of learning setups. Based on jABC [63], an environment for model-based development of software, LearnLib Studio makes components included in LearnLib available as easy to use graphical building blocks (known as *SIBs* in jABC nomenclature), from which complete learning setups can be created.

The rationale behind developing a model-based construction approach for learning setups is that in real-life situations, the learning procedure cannot in general be considered as a standalone entity. Quite contrarily, active automata learning exists in an entanglement with surrounding systems, most notably, of course, the SUL. In cases where several SUL instances are used to allow for parallelized processing of learning queries, the management of these systems becomes its very own concern that has to be handled. Also, e.g., when employing automata learning as part of regression testing during software development, the repeated execution and evaluation of learning setups has to be accomplished as well.

Service-oriented orchestration, supported by jABC, can be employed to solve these management issues. As such, it makes sense to make LearnLib components available as service-oriented building blocks as well, which is the main premise of LearnLib Studio.

3.2.1. The LearnLib Studio modeling approach

In the literature active learning algorithms are described as entities that interrogate oracles to satisfy arising data needs. It is most straightforward to implement learning algorithms according to this view: once the learning setup is instantiated, control is given to the core learning algorithm, which defers generated queries to an oracle.

This approach can easily be transferred into a modeling view: the modeled learning setup first instantiates and configures all involved components, and then starts the actual learning procedure, which proceeds to form a hypothesis. In this view, the interrogation of the target system is done implicitly according to the setup chosen in the configuration part of the model, and not modeled explicitly. A corresponding modeling approach was employed in

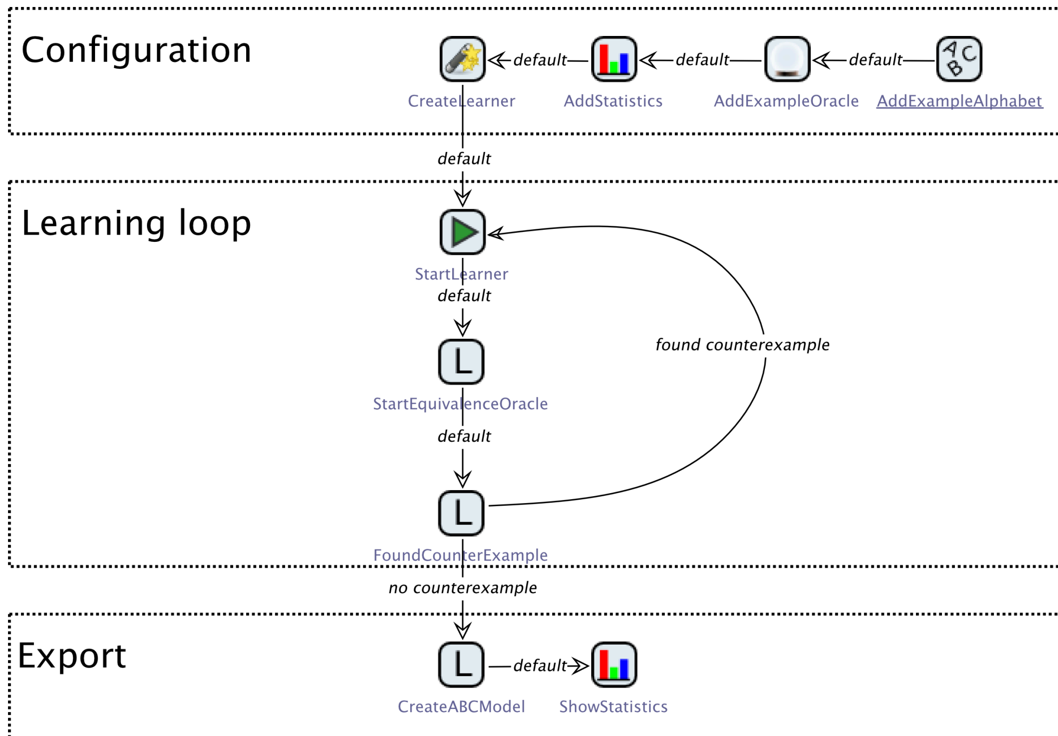


Figure 3.4.: Modeled learning setup with configuration phase and implicit query handling.

early versions of LearnLib Studio, which is illustrated in Figure 3.4: the first steps of the learning process are concerned with creating a fixed setup ("Configuration"), not dissimilar to the learning "pipeline" concept shown in Figure 3.2. The setup is subsequently executed ("Learning loop") and the results of the execution phase are subsequently exported ("Export").

Apart from very simple learning setups, this modeling approach leaves much to be desired, as the actual interrogation is done in seclusion, accomplished in an opaque way according to the fixed preset configuration. This precludes, for example, the ability to model per-case reactions to individual learning queries.

Due to the limitations of the original modeling approach, LearnLib Studio was outfitted with a new modeling paradigm, first described in Paper II. As can be seen in Figure 3.5, the strict phase-oriented structure of the original approach was abandoned. Instead of a configuration phase it is, e.g., sufficient to specify a learning alphabet, after which a learning algorithm is engaged. The queries produced by the learning algorithm are processed individually, enabling fine-grained handling of queries on a per-case basis, as is the case in Figure 3.5, where the number of queries for the target system is decreased by first effecting a cache lookup before eventually asking the SUL for an answer in case of a cache miss.

For being able to implement such a modeling paradigm, the model has to be in control at all times. This means that, e.g., the model has to control when queries are generated and also has to describe how these queries are answered. This, at first, seems to clash with the conventional view onto active learning, where the core learning algorithm is in charge of emitting queries without external control. However, it is possible to execute the core learning algorithm within its own control flow (e.g., by means of multithreading) and have

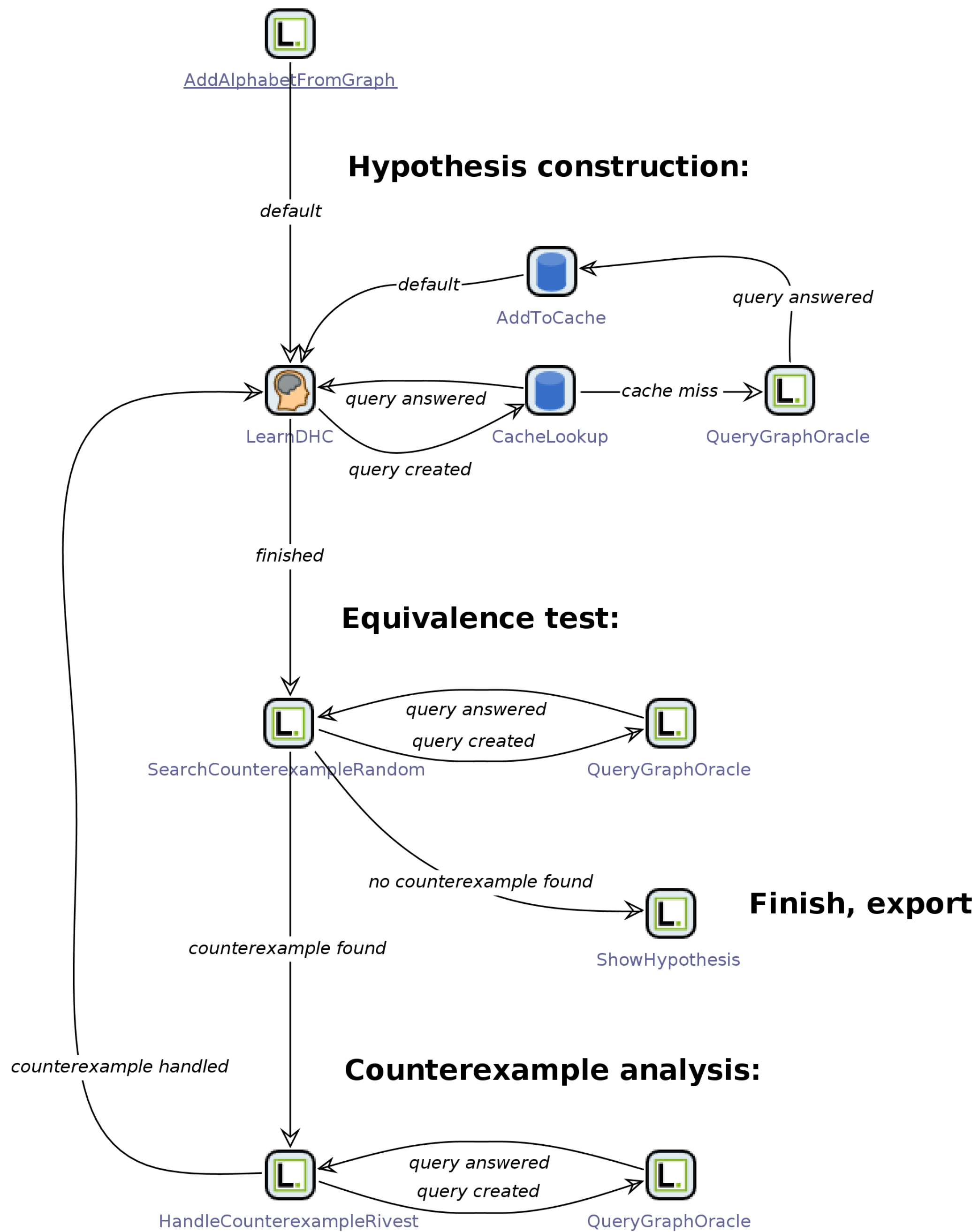


Figure 3.5.: Modeled learning setup utilizing the new modeling paradigm.

the generated queries buffered in a “query pool”. The model of the learning setup then can retrieve queries whenever called for, and proceed to process them freely according to a modeled strategy, with system interrogation being fully transparent at model-level.

3.2.2. SIB learning

LearnLib Studio is not restricted to merely modeling learning setups. Being based on jABC technology, it can be outfitted with additional plugins and SIB libraries to control arbitrary systems. For instance, by employing a specialized SIB palette web applications can be controlled by process graphs.

In case of web applications, specialized browser plugins can record user interactions and subsequently translate those into process graphs composed of individual steps, i.e., a sequence of fittingly parameterized SIBs. Coupled with the execution facilities of jABC and surrounding management processes, powerful test suites can be assembled and orchestrated, e.g., for regression testing.

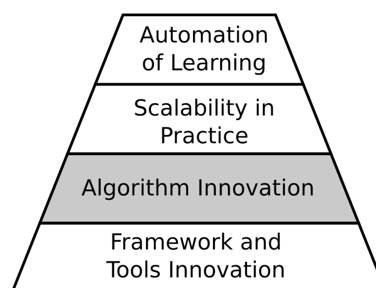
Once a suite of test cases is assembled, LearnLib Studio allows for generating automata models. This can be achieved by selecting interaction SIBs or complete test cases and using those as alphabet for active automata learning. The learning procedure then utilizes the execution resources of the jABC to pose membership queries to the target system.

By constructing the learning alphabet from executable SIBs and process graphs, the resulting models themselves are composed of executable building blocks, enabling direct execution. The potential of combining static test suites and dynamically learned models has been demonstrated in Paper VI (which employs an early precursor of LearnLib Studio), while the idea of basing learning alphabets on SIBs has also been explored in [8] and [9].

Apart from being executable, those SIB models can also be subjected to analysis by model checkers such GEAR [7]. With model checking it possible to assert properties such as that, e.g., protected areas within a web applications are only accessible via a login procedure. The learned models can also be used to generate further tests, in the end coming full circle, but with refined documentation of the target system, asserted properties and additional tests.

4. The DHC algorithm

While LearnLib implements a variety of well-known learning algorithms, it also implements a new and unique algorithm which is discussed in detail in Paper III. This new algorithm is easy to follow, separates data structures for observation storage and hypothesis construction, and can generate big batches of learning queries that can be executed in parallel. It emerged during the effort of rethinking the overall structure of learning procedures that lead to the reengineering of the LearnLib framework (as described in Chapter 3) and was first implemented within the freshly laid out component model.



4.1. The core of the algorithm

In contrast to the L^* algorithm, the DHC algorithm does not employ an observation table to organize information gathered on the target system and from which conjectures are formed. Instead, the algorithm directly constructs an automaton hypothesis. Hence the name of this algorithm: Direct Hypothesis Construction (DHC). Hypothesis construction works along the concepts of *completeness* and *output signatures*, which are defined as follows:

Definition 5 *A state is called* complete *if an output symbol is determined for all transitions, i.e., for all symbols of the input alphabet. A hypothesis is complete if all of its states are complete.*

Definition 6 *The ordered set of output symbols produced by a state in response to input symbols is called* output signature*. The output signature of a state is only well-defined once the state is complete.*

The general idea of the DHC algorithm is to explore the target system in a breadth-first manner and to construct a hypothesis model in the process. As states are visited, MQs are composited by concatenation of the states' access sequences in the hypothesis model with symbols from the input alphabet, which serve as suffixes denoting transitions. The output produced by the SUL is annotated at state transitions accordingly. These steps ensure that every visited state is completed and thus possesses a well-defined output signature.

States with equal output signatures, also called *siblings*, are considered to be equivalent and thus merged into a single state by the DHC algorithm. If a freshly explored state is not considered equivalent to a preexisting state (which means that a new system state was

discovered, witnessed by a new output signature), successor states for all transitions are enqueued for future exploration.

The breadth-first nature of the DHC algorithm can be observed in Figure 4.1, which illustrates the basic construction steps according to Example 1 introduced in Section 2.1. The hypothesis at first only consists of the incomplete starting state, which is completed using membership queries. This results in new incomplete states that are completed using additional queries. In this example the leftmost successor of the starting state shows the same output behavior after completion, which results in this state being merged with the starting state.

An overview in the form of pseudocode is provided in Figure 4.3.

In a sense this construction can be related to the construction of a hypothesis from a tree-like data structure discussed in Section 2.2 for passive learning. The notion of state *completeness*, however, avoids the ambiguity problems discussed in the context of passive learning algorithms. Another difference is that folding of the model is done directly during construction, not as a form of postprocessing on gathered observations.

4.2. Algorithm termination

The DHC algorithm will terminate once the queue of states to be explored is empty. This happens when no non-equivalent states are discovered, i.e., if the output signature of every freshly explored state matches a previously discovered state.

Given that the automaton representation of the SUL is guaranteed to only possess a finite number of states (otherwise no finite-state automaton representation is possible), only a limited number of distinct output signatures can exist, guaranteeing algorithm termination.

4.3. Refining hypotheses

The first hypothesis constructed by the DHC algorithm for the coffee machine example is displayed in Figure 4.2. It is easy to see that this model is not a faithful representation of the target system visualized in Figure 2.3. As is the case with all MAT algorithms, the hypotheses constructed by the learning algorithm are subjected to an equality test by means of EQs.

If a counterexample was discovered, one of its suffixes can be identified that reveals diverging behavior for at least one state in the hypothesis model, as mentioned in Section 3.1.2. Rivest and Shapire described a procedure in [60] employing a binary search on the counterexample to pinpoint the relevant suffix (the procedure is also illustrated in Paper I within the coffee machine example). The identified suffix is inserted into the learning algorithm's alphabet. Subsequently construction of a new hypothesis will be started from-scratch with the refined alphabet, which includes the new suffix to reveal diverging behavior between at least two states.

Figure 4.4 demonstrates the effect of adding such a distinguishing suffix. The diverging output for the distinguishing suffix "*water button*" at the states s_0 and s_1 ensures that the output signatures of these states are different. This prevents the merging of those two states, in contrast to what happens without a distinguishing suffix in Figure 4.1, where the output

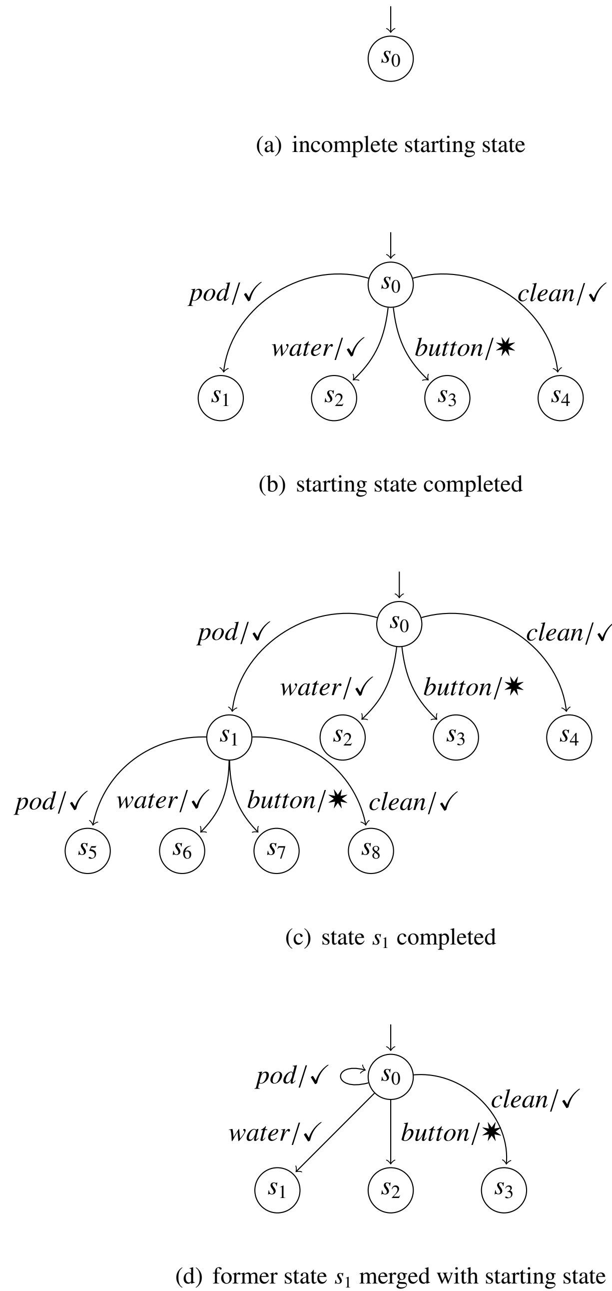


Figure 4.1.: First steps of DHC hypothesis construction. Source: Paper I

Figure 4.2.: First hypothesis constructed by the DHC algorithm.

```
1  function DHC(Alphabet alphabet) {
2    Hypothesis hypo = new Hypothesis();
3    Queue worklist = new Queue();
4    worklist.enqueue(hypo.getStartState());
5
6    while(worklist.isNotEmpty()) {
7      State currentState = worklist.dequeue();
8      Sequence accessSeq = currentState.getAccessSequence();
9      for(Symbol sym in alphabet) {
10       Query query = accessSeq.append(sym);
11
12       // Communicate with the target system, fetch output symbol
13       Symbol output = doMembershipQuery(query);
14
15       // set the transition output for the sym input-symbol
16       // to the retrieved output symbol.
17       currentState.setTransitionOutput(sym, output);
18     }
19
20     State sibling = findOtherStateWithSameSignature(currentState);
21     if(exists(sibling)) {
22       // reroute all transitions to currentState to sibling
23       hypo.rerouteAllTransitions(currentState, sibling);
24       hypo.remove(currentState);
25     } else {
26       currentState.createSuccesorsForEveryTransition();
27       for(State successor of currentState) {
28         worklist.enqueue(successor);
29       }
30     }
31   }
32
33   return hypo;
34 }
```

Figure 4.3.: Pseudocode of the DHC core algorithm. Source: Paper III

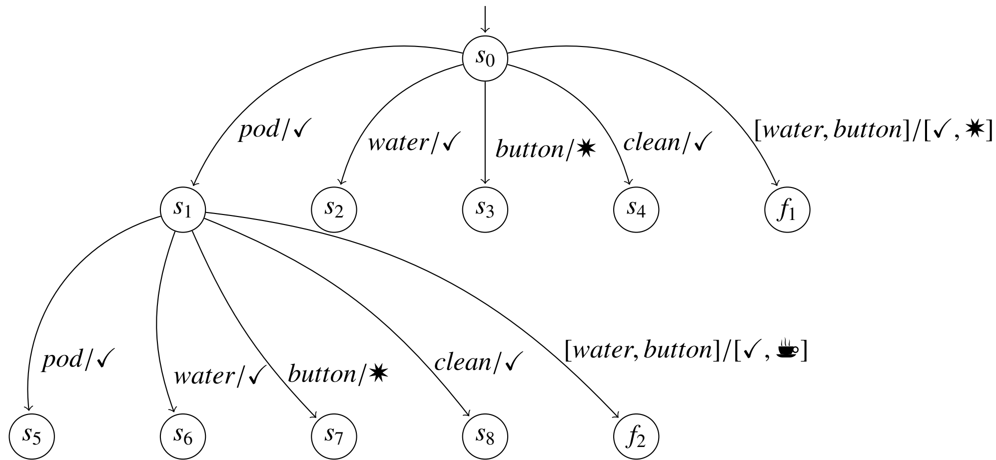


Figure 4.4.: An early stage of the DHC hypothesis construction, corresponding to Figure 4.1(c), with the distinguishing suffix “*water button*”. Source: Paper I

signatures match. Note that in Figure 4.4 states reached by distinguishing suffix transitions are named with a diverging scheme so that the state names in Figure 4.1(c) are also valid in this figure for quick comparison. The letter “f” is used to express that the distinguishing suffixes reveal peeks into *future* behavior. Also note that these states do not need to be enqueued into the breadth-first exploration queue, as they are guaranteed to be visited by the successive execution of the singular input symbols of the distinguishing suffix, necessitating only the exploration of successor states for original input alphabet symbols.

4.4. Memory consumption

The DHC algorithm allows for unparalleled flexibility regarding the memory profile of the learning algorithm. This is due to DHC being the only algorithm that cleanly separates following vital concerns:

- The constructed model steers exploration and also represents the hypothesis at any given time. By its very nature this data structure does not outgrow the minimal representation of the SUL and thus denotes the absolute bare minimum of memory resources needed to successfully generate a model.

  This compact in-memory representation comes, however, at a price. While L^* refines hypotheses by adding new rows and columns to the observation table, keeping all information already gathered intact, the DHC algorithm resumes construction from scratch as part of counterexample handling. This means that each refinement iteration will repeat every single membership query from the previous iteration, with new queries being generated additionally (caused by the new alphabet symbols). Depending on the time needed by the teacher to answer one MQ, this can add considerably to the overall time consumed by the learning procedure, if no data structure for observation storage (i.e., a cache) is employed to filter out repeated queries.

Storage technology	Access times	Typical capacity
Hard Disk Drives (HDD)	10^{-2} s	2000 GB
Solid-state Drives (SSD)	10^{-4} s	200 GB
Dynamic Random Access Memory (DRAM)	10^{-8} s	8 GB

Table 4.1.: Approximate access times and storage capacities for various storage technologies. Typical storage capacities are rough estimates for well-equipped desktop PCs (September 2012).

- A data structure is utilized to organize future exploration, i.e., a work list. For the DHC algorithm, being inspired by classical breadth-first search, this is a simple queue of yet unexplored states. Given that these unexplored states can only be children to freshly discovered states, a memory conserving alternative to directly enqueuing these unexplored states is to enqueue only the parent state. Subsequently, the children that originally would have been enqueued can be recovered by iterating over the alphabet. This means that the work list can be implemented in a way so that memory consumption is bounded by the size of the target system.
- The data structure associating discovered states to their respective output signatures can easily be implemented as a decision tree. During learning, the output signature cannot reasonably gain more symbols than states in the final hypothesis, meaning that worst-case memory consumption of this structure can be estimated to be dominated by $O(n(n+1)/2)$, while in practice the memory consumption seems to be much more benign as distinguishing suffixes usually result in more than one state being split. Being subject to regular access patterns, the decision tree can be organized for mass storage devices in cases where this data structure grows in a less good-natured way.
- Observation storage can either be omitted (in cases where repeated membership queries are inexpensive) or be delegated to a cache, which can either reside in system memory or be externalized. Again, a decision tree can be used as underlying data-structure, which can easily be maintained on external storage.

In summary, DHC offers unique possibilities to optimize the memory profile for the application in mind, balancing speed and storage needs. This is achieved by clear separation of concerns between data structures, whose simplified data access patterns enables easier externalization than a "one structure fits all" observation table. As presented in Figure 4.5, the DHC algorithm can achieve memory consumption scaling with the number of discovered states, which compares favorably to L^*.

Modern computer systems employ a hierarchy of different storage technologies. As a rule of thumb fast storage technologies (such as DRAM) are expensive and thus in practice limited in capacity, while cheap storage technologies (such as HDDs) can offer abundant capacity. As illustrated in Table 4.1, the differences in access times between fast and slow storage span several orders of magnitude, as is the case for the typical storage capacities.

By separating the various storage concerns into distinct data structures, the DHC algorithm makes it possible to make use of the various levels of the memory hierarchy according to the respective properties. For instance, the constructed hypothesis is rather compact and

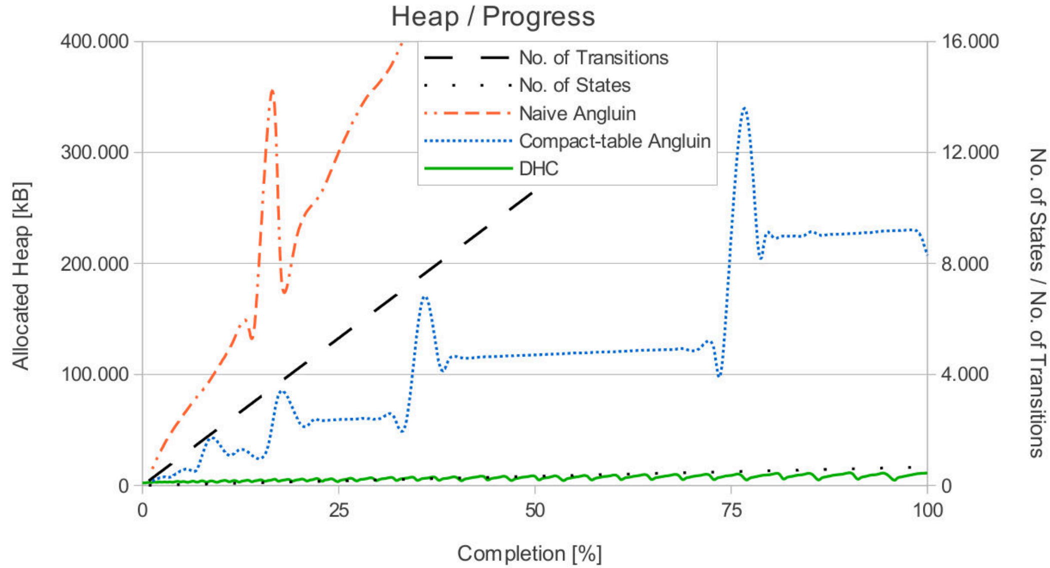


Figure 4.5.: Memory consumption of the DHC algorithm compared to two L^* implementations. Source: Paper III

is accessed and modified rapidly by the learning algorithm. Observation storage, however, may consume considerable storage space, but is less performance critical as even an externalized data structure for observation storage on a (comparatively slow) hard drive can be significantly faster than actually querying the SUL. The other data structures fall in-between those two extremes regarding access frequency and size and thus can be fitted into the memory hierarchy according to the resources at hand.

4.5. Number of generated MQs

In Paper I the worst case number of MQs generated by DHC during a complete learning run is given as $O(n^3mk + n^2k^2)$, where n is the number of states, k the size of the original alphabet Σ, and m the length of the longest counterexample. This assumes that all suffixes of a counterexample are added to the learner's alphabet instead of just one suffix per counterexample as discussed in Section 4.3. Another assumption is that no observation storage is used to filter out repeated queries.

When analyzing an optimized version, following examinations can be made for the last DHC iteration, i.e., the iteration that yields an adequate system model: The size of the learning alphabet cannot exceed $k+n$, as there can only be n counterexamples, each yielding one additional distinguishing suffix. The learning algorithm cannot discover more than n unique states, each necessitating $k + n$ queries for completion. As it suffices to explore only successors reached by symbols of the original alphabet S, each unique state has k successors that need to be completed with $k + n$ queries. This means that the worst-case number of membership queries generated by the last iteration is $n(k + n) + nk(k + n)$, which is $O(n^2k + nk^2)$. This is also the number of MQs that the SUL has to process if a data structure for observation storage (i.e., a cache) is employed to filter out repeated queries,

otherwise the number of queries can be estimated to be $O(n^3k + n^2k^2)$, as there are at most n iterations.

Analysis of counterexamples by means of binary search consumes $\log(m)$ MQs per counterexample, meaning that a complete learning setup utilizing the DHC algorithm with observation storage will consume $O(n^2k + nk^2 + n\log(m))$ queries. This is the same estimate as provided for an optimized L^* algorithm in Paper I, indicating that the DHC algorithm can be configured to be as efficient regarding MQ numbers as optimized conventional algorithms.

4.6. Creating query batches

As described in Section 3.1.3, LearnLib supports processing several membership queries at once to optimize filter efficiency or to distribute queries within a networked cluster of SUL instances.

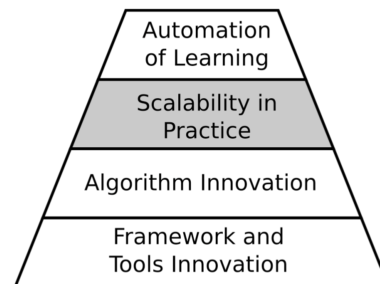
To take advantage of this mode of operation it is necessary that the employed learning algorithm and its concrete implementation within the LearnLib framework support creating such queries. Furthermore, to make best use of this mode of operation it is desirable that batches contain as many queries as possible, e.g., to be able to assign at least one work item per network node.

Generating batches is exceptionally easy for the DHC algorithm, as at any time it is possible to generate queries for all enqueued states in the work list. For comparison, the L^* algorithm is structured in phases of resolving unclosedness and inconsistencies, which limit the potential for assembling learning queries. The efficiency of various learning algorithms in a parallel and distributed environment will be discussed in more detail in Section 5.2.

5. Scalability in practice

Even with the availability of flexible and well-structured learning tools, applying automata learning techniques to real life systems presents challenges which, however, can often be overcome by means of careful evaluation and adaptation of the learning setup. Some of these challenges concern time constraints (e.g., learning taking too much time), resource constraints (e.g., not enough computing resources being available) or constraints regarding the fitness of the target system for automata learning (ensuring determinism and observational independence of learning queries).

In this chapter an overview of the most profound challenges will be provided, alongside strategies for solving the respective problems. Thus this chapter builds on top of the innovations regarding the learning framework and algorithms.



5.1. Practical concerns

Regardless of the concrete implementation of active automata learning, all learning setups share common practical concerns which have to be addressed for successful application of automata learning techniques to real-life examples. In the following some of these concerns will be discussed, alongside sketches of how to address them utilizing the infrastructure LearnLib does offer.

5.1.1. Challenges for active automata learning

Even through creating learning setups with LearnLib is easy, successful operation of any of these setups also depends on "external" factors, many of those regarding the SUL. Thus challenges remain that have to be overcome before reliable or comprehensive results can be expected:

1. **Determinism:** Active automata learning in the sense of Angluin deals with deterministic systems, i.e., that every membership query will return the same result when repeated. When dealing with reactive systems this means that the same output has to be produced whenever the same inputs are applied.

   It is important to keep in mind that from the point of view of the learner, the test driver and the SUL are indistinguishable. This means that eventual bugs in the test driver can introduce problematic non-determinism. The upside, however, is that the test driver is in the position to produce a deterministic view of an eventually non-deterministic system. One simple example are timestamps in output messages that

can be removed by the test driver before submitting the query result to the learner. In this case, the test driver is applying an abstraction step to the output produced by the SUL. However, this can just as well be accomplished by a dedicated abstraction facility that is inserted into the `MembershipOracle` chain depicted in Figure 3.2. In a similar fashion a test driver can preserve determinism by detecting spurious errors of the SUL and determining a “correct” result by repeated execution.

2. **Reset:** The determinism requirement implies that all queries have to be executed independently from one another – the outcome of succeeding queries may not be influenced by their predecessors. One way to achieve this is to perform a reset procedure on the target system, which transfers it into a well-defined starting state.

   Performing an actual system reset is, at times, challenging. For example, a “target system” may indeed consist of several networked nodes, each of which has to be included into the reset procedure. Even if it possible to consider all involved nodes, it may be non-trivial to identify the actual subsystems that contribute to the overall system’s observable state: for example, in many web applications the system state is solely held in a central database, which means that one can force the system into a well-defined initial state by importing a previously taken database snapshot. Applications deployed in a business application server, however, are also influenced by the state of the container they are executed in.

   On a closer look, all that is needed to satisfy the “reset” requirement is that queries produce no side effects on each another. In practice, many multiuser systems are designed to service users independently, e.g., by employing a session concept and containing interaction effects within one session. In these scenarios, all that is needed to perform a “reset” is to open a new session. This, again, can be seen as a form of abstraction, where the “abstract” need of observationally independent queries is concretized into independent sessions on the SUL. This approach of doing reset by abstraction was first proposed in [1].

3. **Execution time:** During automata learning, depending on the size of the target system, several hundred thousand membership queries may be needed to conclude with well-formed hypothesis, each query involving actual execution on the system to be learned and execution of a reset procedure. When considering that each system invocation and system reset may consume several seconds, the dimension of this problem becomes obvious. Once again referring to Figure 3.2, the query cache and the `MembershipOracle` instances implementing application-specific filters aim at answering learning queries without system invocations, saving on execution time.

These concerns can be accounted for by developing application-fit query processors, i.e., a `MembershipOracle` component that encapsulate application-specific aspects. For instance, SUL resets, either actual system resets or abstracted ones, are realized by an application-fit implementation of the `MembershipOracle` interface. As already indicated, the execution time can be decreased by filters that reduce the overall amount of queries that have to be processed by the SUL. These filters, again, are `MembershipOracle` implementations.

5.1.2. Filter example: The query cache

A query cache is a particularly useful filter for many learning setups. If, for example, the DHC algorithm is used, the query cache will filter any queries the learning algorithm asks repeatedly as part of its hypothesis refinement working principle. Given how important a query cache is for the efficiency of the DHC algorithm, the implementation which is now available as a reusable component originally started as an internal data structure of LearnLib's DHC realization.

However, it was quickly realized that this component is useful in other contexts as well, which is why the data structure was made generally available. For instance, the cache can answer all queries without further system interaction that are prefixes to already processed queries. This is especially useful for learning algorithms that can assemble query batches, as those can be sorted accordingly.

Another useful property of the cache is that it can detect non-deterministic behavior of the SUL: as the cache memorizes all queries and their respective outputs, every new query can be checked against shared prefixes of already stored queries. If the output for any prefix differs from the newly retrieved output, this means that the SUL provided inconsistent answers, e.g., because of intrinsic nondeterminism or because of an incorrect reset procedure in the test driver. The cache then can signal an appropriate exception, including the conflicting query outputs for debugging purposes.

The query cache implements the `MembershipOracle` interface and operates as part of a chain of components: the cache either can provide an answer for an incoming query, or simply delegate the query to the next oracle in the chain. Once an answer for the delegated query arrives, the cache will be updated accordingly.

A suitable data structure for the cache is a decision tree, with nodes that can store output symbols and which are connected by edges associated with input symbols. In LearnLib, two implementations exist: one that resides in system memory, and an externalized version that is stored on hard drives.

5.2. Distributed learning

Although query filters can significantly reduce the number of membership queries that have to be processed by the SUL [36, 46, 8, 38], for complex systems a high residue of necessary queries remains. Depending on the speed in which the target system can produce output, the overall time needed to sequentially process queries can easily become unacceptable.

A possible solution is to create several instances of the SUL, and process several queries in parallel. For this to work, however, the employed learning algorithm has to generate several independent queries at once and provide them in batches, as discussed in Section 3.1.3. For maximum effect, the batches have to be big enough as to provide queries for all SUL instances, meaning that big batch sizes are desirable.

In Paper IV the potential the parallelization potential of different learning algorithms is investigated. As can be seen in Figure 5.1, the size of batches varies substantially between algorithms. While the DHC algorithm can generate the biggest batches, their size can fluctuate wildly. The L_M^* algorithm has a more even distribution, but on average creates smaller

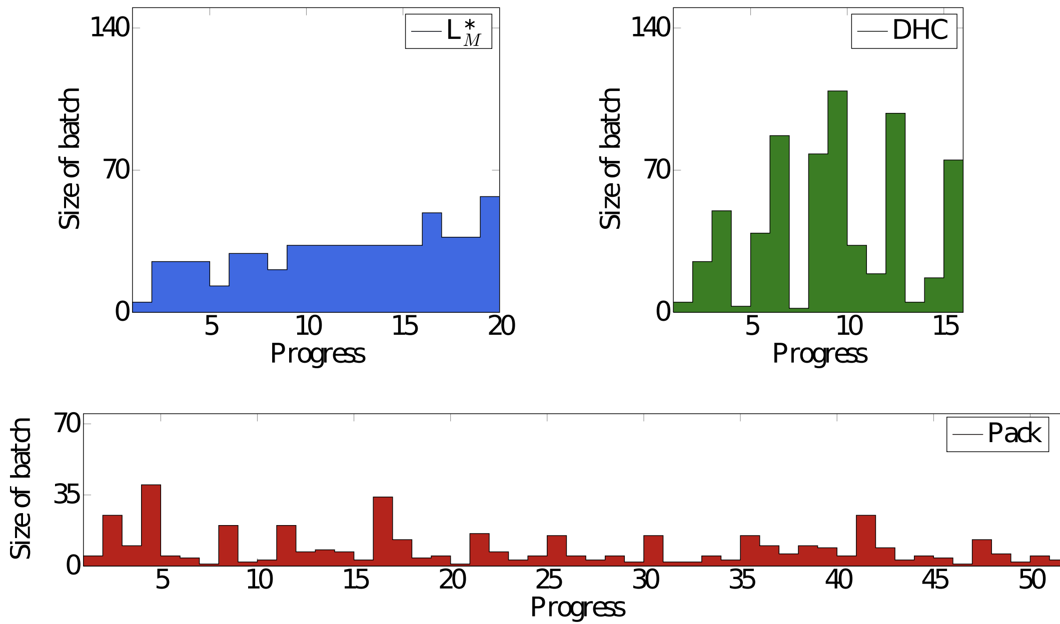


Figure 5.1.: Batch sizes for different algorithms implemented in LearnLib. Source: Paper IV

batches. While these two algorithms appear to be a good fit for parallelized processing of queries, the Packs algorithm [32] appears to be much more limited in this regard.

In Figure 5.2 experimental results are provided for a random system with 15 states and 4 input symbols. The experimental setup simulates a scenario where each query is answered with a delay of 500 ms, which experience shows can easily be reached or exceeded when dealing with real-life systems. Repeated execution of the experiment revealed that the results obtained are indeed stable over the course of several runs.

With few threads, i.e., few queries being answered in parallel, the Packs algorithm takes less time to learn a model than the other algorithms included in this comparison. This can be attributed to the fact that the Packs algorithm usually generates fewer membership queries and instead issues more equivalence queries compared to the other learning methods, which favors the Packs algorithm as equivalence queries immediately return an accurate answer in this particular experimental setup. Nonetheless, just like the data presented in Figure 5.1 suggests, the Packs algorithm does not scale well with increasing parallelism. In contrast, both the L^*_M algorithm and the DHC algorithm scale much more gracefully as they emit bigger batches of queries. Overall, the DHC algorithm scales best and eventually pulls noticeably ahead of the L^*_M algorithm. However, while increasing the thread count gives near-linear speed increases up to approximately 16 threads in this experiment, subsequent speedup is much tamer and eventually completely vanishes. Referring again to Figure 5.1, this can be explained by the non-uniform batch sizes generated by the learning algorithms, meaning not every batch can take advantage of the added processing resources, leaving threads idle. Nonetheless, overall, significant time savings can be achieved by processing queries in parallel, resources permitting.

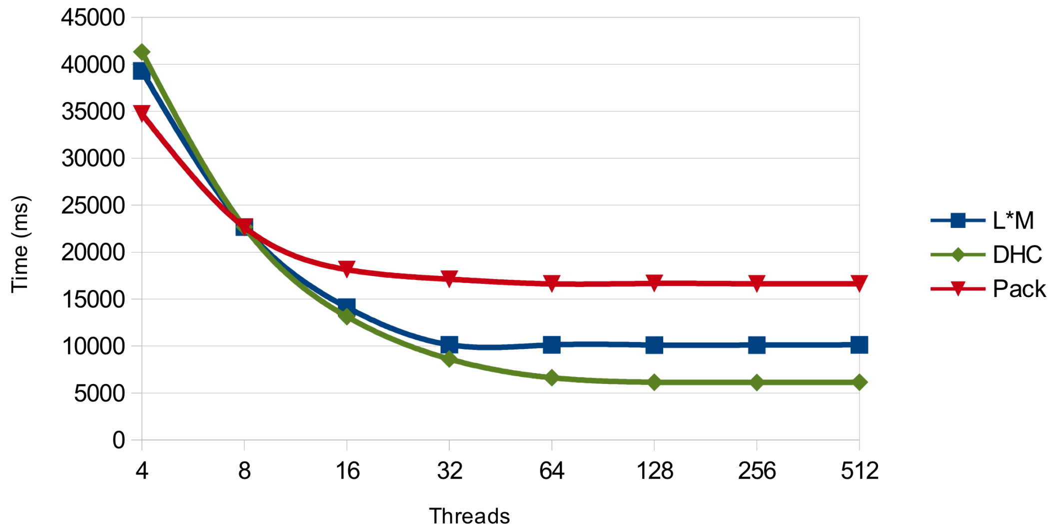


Figure 5.2.: Scaling of different learning algorithms in a simulated distributed setup.

5.3. Moving learning into the cloud

As has been described in the previous sections, utilizing networked systems for automata learning in the form of, e.g., parallelized and distributed processing of queries can significantly accelerate automata learning. Beyond processing queries in a local private cluster of networked computers, it is also possible to move all concerns of automata learning, from configuration to execution, onto remote systems. Instead of provisioning local resources, which can cause considerable expenses, cloud computing can provide such resources as needed.

The following sections provide a preview on how cloud computing may be utilized to provide application-fit learning setups on demand.

5.3.1. Cloud computing and automata learning

Comprehensive learning of real-life systems can bind considerable resources:

- Depending on the size of the target system, thousands or hundreds of thousands of queries are directed at the target system. To generate a result within an acceptable time-frame, the target system should be able to process several queries per second on average. This means that adequate resources have to be provided so that the interrogated system can operate in a timely fashion. If several SUL instances are to be used to process queries in parallel as described in Section 5.2, even more resources have to be made available accordingly.

- The learning algorithm and related components can consume considerable resources as well. For instance, observation tables such as employed in Angluin's L^* algorithm can quickly consume several gigabytes of system memory, which can limit the size of the models that can be learned in practice.

In effect this means that employing active automata learning to explore real-life systems can require significant investments to assemble a setup that can meet all requirements. One potential solution to cater for these demands is cloud computing [50].

The main idea of cloud computing is to access resources over a high-speed network as needed, delegating the need to provide and maintain computing infrastructure and services to specialized providers that can operate cost-efficiently. This concept can provide the following advantages:

- The consumer only pays for those resources that actually have been consumed. The provider, on the other hand, can even out utilization rates of computing resources due to a high number of consumers.
- The cost structure is known beforehand to the customer and consumed resources can usually be monitored in near-realtime. Thus the customer is protected from hidden costs that can occur when provisioning and operating dedicated resources.
- A single consumer usually only consumes a fraction of the overall resources that can be provided. Thus it is possible to rapidly react to utilization changes and scale as needed.
- Cloud services are in general hosted in dedicated data centers. Many big providers of cloud services operate several geographically distributed data-centers so that localized failures can be compensated by migrating services to another location. This can increase reliability.

The number of services that can be provided by cloud-computing can be manifold, but most services can be attributed to one of the following categories:

- Infrastructure as a Service (Iaas): Resources such as computing time and storage space are provided and accounted for.
- Platform as a Service (PaaS): A software framework is provided that enables access to cloud resources. Applications that are to be deployed into the cloud have to utilize this framework to use cloud resources.
- Software as a Service (SaaS): Complete applications are provided to the consumer. Software maintenance and operation is completely delegated to the cloud provider.

Complete learning setups can easily be instantiated and executed on cloud platforms. A simple scenario that does not necessitate any software changes is the remote deployment of virtual machines encapsulating both the learning setup and the SUL. However, such setups that merely emulate conventional computers do not automatically allocate additional cloud resources when the need arises. This motivates the development of solutions that are "native" to the cloud environment and allow for flexible management of cloud resources. In the following, the WebABC, a SaaS solution for creating and executing modeled learning setups in a cloud environment, is presented.

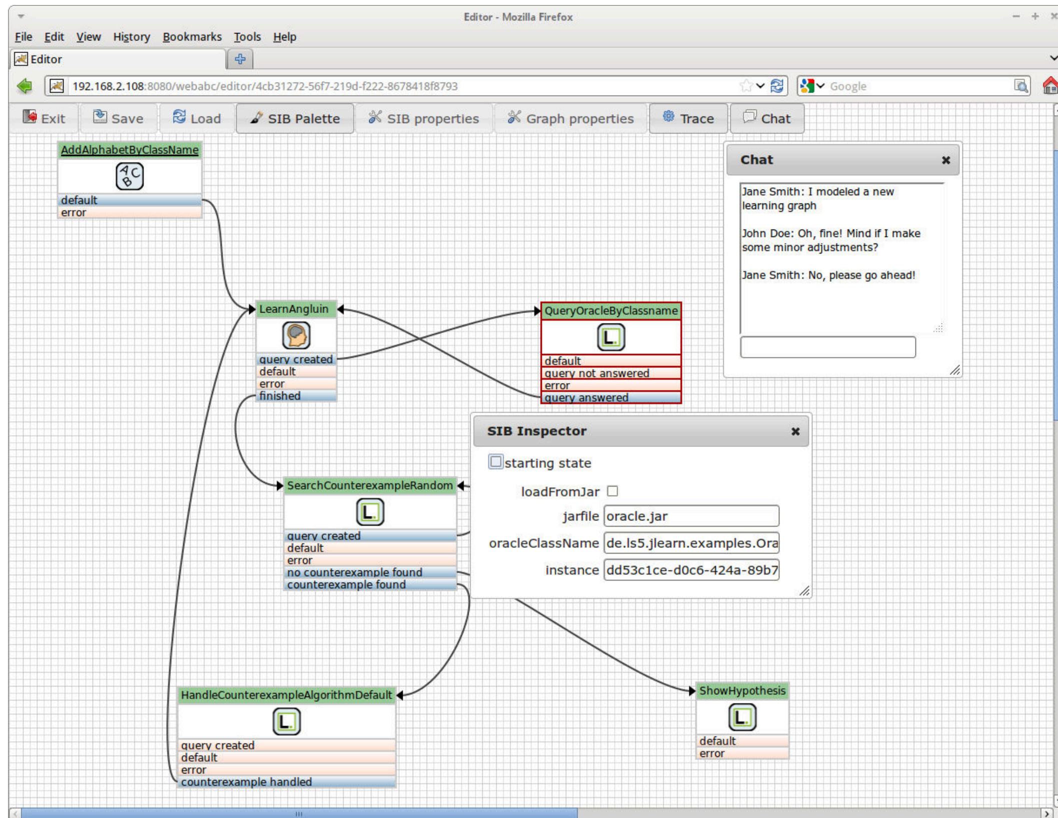


Figure 5.3.: User interface of WebABC, with a modeled learning setup visible.

5.3.2. The WebABC

Acquiring necessary resources for learning setup execution from cloud vendors instead of undertaking local provisioning marks a transition from localized learning setups into a highly networked environment with maximum flexibility and cost transparency. Beyond the mere execution of learning procedures, owing to advancements in web technology, a completely cloud-based environment, including a rich user interface for creating setups, becomes increasingly feasible.

To explore this potential, the WebABC, a web-based re-envisioning of the jABC, was developed. Currently existing as a prototype, it allows for modeling and executing learning setups and thus can be seen as web based version of LearnLib Studio. Figure 5.3 provides a glimpse at what modeling learning solution in WebABC looks like. As can be seen, most screen space is dedicated to the drawing canvas, with additional UI elements such as the user chat and SIB parameter inspector being displayed as windows that can be rearranged according to the user's needs.

The main rationales for moving the complete infrastructure, including the graphical user interface and the execution layer, into a cloud environment, are the following:

- Moving into a distributed environment with parallelized processing of queries can induce considerable management needs for the overall setup. For instance, if several SUL instances are to be employed, the life cycle of each instance has to be controlled in accordance to the needs of the learning setup. If, e.g., virtual machines containing

the SUL are to be used, the setup management has to make sure fitting virtual machine images are distributed, started, subjected to reset procedures as queries are processed, and finally terminated once learning finishes. The most straightforward approach to realize this is to execute the management process in close proximity to the managed entities, i.e., within the cloud environment.

- The execution layer can dynamically scale the allocation of cloud resources according to user needs and, e.g., replace faulty processing nodes, thereby increasing reliability.
- Providing the user interface as SaaS cloud service provides the users with a ready-to-go solution, without the need and sophistication of configuring and maintaining a local setup. In addition, due to the networked nature of the SaaS solution, features such as collaborative editing and messaging between users can easily be made available.

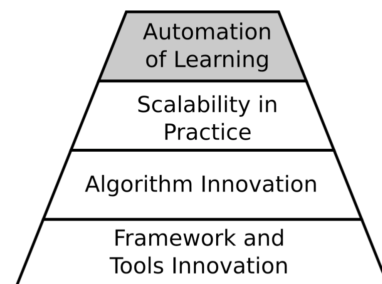
The WebABC demonstrates the feasibility of providing a rich graphical editing environment in a SaaS fashion, coupled with a server-side execution layer that is prepared for extension into the realm of distributed cloud execution.

6. Automated configuration of learning setups

A major obstacle for deployment of active automata learning in real-life contexts is the effort needed to design and implement application-fit learning setups. In [62], construction of the learning setup is estimated to have accounted for approximately 27% of the total effort when learning an embedded system.

Creating a learning setup involves determining a suitable form of abstraction and finding ways to manage runtime data influencing the behavior of the target system. While the former has influence on the expressiveness of the finished learned model, the latter is an immediate concern when interacting with reactive systems, where communication often is dependent on concrete data values previously transferred. For example, a system guarded by an authorization system may transport a security token to the client on login, which subsequently has to be included with any interaction with protected system areas.

The learning setup has to translate abstract learning queries into concrete requests to the SUL, which additionally may have to be augmented with data values. In automata learning, the building block facilitating the translation is known as *mapper*. To allow for a high degree of automation in automata learning of reactive systems, the ability to automatically configure this component is a basic prerequisite. In Paper V an approach is presented to configure a reusable test driver with a configurable mapper automatically, with key points being recalled in the following.



6.1. The role of test drivers in active automata learning

In active automata learning, the learning algorithm usually employs an abstracted learning alphabet, i.e., the symbols used are not directly employable as system input for the SUL. Thus, some component in the learning setup has to translate from the abstract realm of learning symbols into the realm of concrete stimuli for the target system. To actually be able to observe output generated by the reactive system under observation, these concrete stimuli also have to be fed into the SUL, using whatever means available to interact.

In current practice, hand-crafted test drivers are employed to attend to this task, using hardcoded translation mechanisms that facilitate the abstraction and concretization steps, in addition to effecting system interaction and output observation via system-specific interfaces. Overall, these manually created constructs do not lend themselves to reuse, leading to ad-hoc solutions with very limited scopes of application.

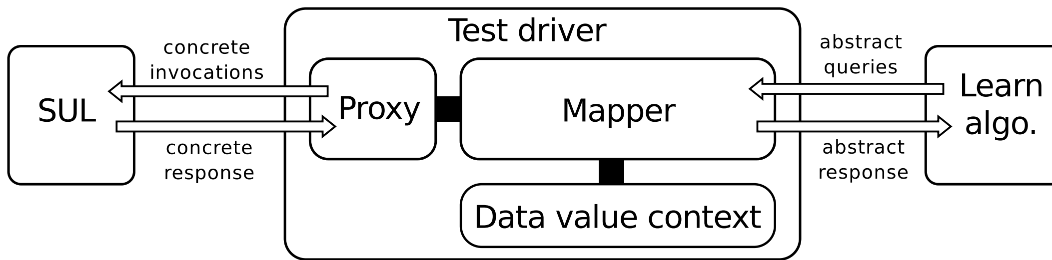


Figure 6.1.: Architecture of a configurable test driver for active automata learning. Source: Paper V

To alleviate this problem a configurable – and thus reusable – test driver has been developed for LearnLib, which is illustrated in Figure 6.1.

Following components are visible within the architectural overview:

- A *mapper* facilitates the translation of abstract learning symbols into concrete input symbols for the target system. For parameterized abstract input symbols, the mapper determines fitting runtime values when creating concrete system input. The mapper is also responsible for abstracting from concrete output values of the SUL into an abstract output alphabet. Mappers are discussed, e.g., in [41].

- The *data value context* contains runtime data values and predefined values such as, e.g., credentials, which have to be provided beforehand. The data value context provides concrete data values to the mapper to fill out symbol parameters. When abstracting from runtime output values, the mapper will submit these values to the data value context for future reference.

- The *proxy* is an exchangeable component that offers a unified invocation interface to the SUL and returns system output. It maintains a connection to the target system and allows the test driver to direct queries to the SUL. In case of systems with a Java interface, the proxy can interact with the system under examination by employing the Java class reflection API [55], which offers a standardized way to invoke methods. Given that many target systems have interface descriptions in standardized formats such as WSDL (Web Services Description Language, for web services [66]) or IDL (Interface Description Language, for CORBA services [54]) that can be converted into Java interfaces, this offers a unified means for target invocation.

In the reusable test driver, all three components merely need system-specific configuration instead of per-system modification of the test driver code. As will be sketched in the following, for many systems this configuration can be derived by means of interface analysis, paving the way for learning setups with a high degree of automation.

6.2. Learning setup creation by interface analysis

Fitting configurations have to be determined for the mapper, the proxy and the data value context. This means that the alphabet has to be determined, a means to interact with the system has to be employed, and a strategy how to deal with runtime data values has to

be found. These concerns can be addressed by interface analysis, as demonstrated in the following.

6.2.1. Determining a learning alphabet

For many systems, the API provided by the system's developers is a natural source for the learning alphabet: most APIs are designed to encapsulate concrete system actions behind an abstract interface, shielding the user from the exact technical implementation. In fact, this encapsulation of implementation-specific details is a main motivation for defining APIs. Beyond this encapsulation aspect, API methods also are often concerned with offering concise ways to effect isolated use cases, being structured from a point of view of a user.

This allows for a straightforward way of determining a learning alphabet, which reflects the abstract view onto the target system as offered by its API:

- For each *method of* the API an abstract learning symbol is defined, which can, for instance, aptly be named after the corresponding method. At runtime the mapper component within the test driver will issue an actual method call whenever encountering an abstract symbol.

- Each *parameter* of an API method is taken into account by parameterizing the corresponding abstract learning symbol. At runtime the data value context will be queried for live data values to fill out these parameters for actual system invocation.

- *Return values* (i.e., system output) are converted into abstract output symbols by merely indicating if the corresponding invocation was successful (and thus returned a data value) or did lead to a system error (for instance, a system exception occurring). The latter case can be signaled by emitting an abstract "error" symbol, the former by emitting an output symbol named after the data type of the returned data value. Any returned data value should be submitted to the data value context for future reference.

The necessary analysis steps can be conducted via the Java class reflection mechanism in case of Java interfaces. Alternatively, for interface descriptions in WSDL or IDL specialized parsing tools can be employed. However, tools such as `wsimport` and `idlj` (which are part of the standard Java Development Kit) can convert these interface descriptions into Java interfaces, making them accessible to the very same analysis mechanism.

6.2.2. Interfacing with the target system

Invoking systems with a well-defined API is generally a trivial task: in case of Java interfaces, invocation is usually possible without much preparation by a generic proxy component that utilizes the Java reflection API. For cross-platform interface descriptions such as IDL or WSDL specialized tools can be employed that generate ready-to-use Java classes that offer means for system invocation, hiding the details of the concrete remote invocation mechanism. A generic proxy component can then operate on these generated classes by invoking methods according to their respective names as specified in the configuration, without manual modification of the proxy itself.

6.2.3. Dealing with live data values

To be able to invoke parameterized interface methods, fitting runtime values have to be provided. In many cases, return values provided by one method are needed as input values for another method. This is, for instance, the case for many authorization schemes, where one method will generate and return an authentication token which has to be included with every subsequent invocation. Such dependencies can be observed in the e-commerce scenario discussed in Paper V, where the following methods are exposed via a WSDL interface:

- The `openSession` method expects authentication information and returns a session token associated with a newly created session. Conversely, the `destroySession` methods expects a session token and invalidates the associated session.
- `getAvailableProducts` returns a list of available products. This method has no parameters.
- `addProductToShoppingCart` adds a specified product to the shopping cart associated to a provided session token. The `emptyShoppingCart` method removes all items from the shopping cart of the specified session, while `getShoppingCart` returns an object containing references to all contents of the cart.
- The `buyProductsInShoppingCart` method purchases all items contained in the shopping cart associated with the provided session token.

Data dependencies between these methods have to be determined automatically to enable learning setups without manual intervention. This can be accomplished, e.g., in the following ways:

- If the analyzed interface is built on top of a type hierarchy, the inter-method data dependencies can only exist in alignment with the type system, which means that only those return values need to be considered as possible data dependencies that match the corresponding parameter type. For instance, in the e-commerce scenario of Paper V, the method that adds objects typed as "Product" to a shopping cart has a data dependency on the method that provides a list of available products.

  If the interface makes strong use of types, most dependencies between methods can be ruled out merely by the type concept. Many real-life web services, however, employ a relaxed type concept, where data values are always encoded, for instance, as character strings. In such a scenario of a "depleted" type concept, any return value could fit as nearly any parameter as far as the type concept is concerned. Thus, type analysis alone will give only an inflated approximation of the actual data dependencies, which makes dedicated handling necessary for these cases.
- In cases where no type system exists or is not used in alignment with the inter-method data dependencies (such as in the depleted type system sketched above), a training phase can replace or supplant type analysis. During training actual system invocations are used to confirm or dismiss potential dependencies. While type analysis on a relaxed type system will usually determine more data dependencies than actually exist, this type of training can, however, dismiss real data dependencies that only

occur depending on the current system state. Referring again to the e-commerce example, adding items from the list of available products to the shopping cart is only possible once the user is logged in, meaning that training can spuriously dismiss this dependency if the authorization procedure is not completed beforehand.

Such a training phase can be performed, e.g., by the tool “Strawberry” [10] on WSDL interfaces.

- Given that both type analysis and a dependency training phase may give unsatisfactory results depending on the employed type concept or the thoroughness of testing, another possibility is to evaluate semantic annotations that describe the usage of parameters and return values. In this scenario, type analysis is replaced by semantic reasoning over an ontology that contains concepts the parameters and return values are annotated with. The SAWSDL (Semantic Annotation for WSDL, [23]) specification describes an extension to WSDL which makes semantic reasoning over the individual parts of the interface description possible. Sadly, in real-life, only few (if any) systems expose such a semantically annotated interface.

Once the data dependencies have been determined, one easy way to pass data between method calls is to allocate one variable per data type in the data value context. Parameter values can be filled in by referring to the variable corresponding to the parameter type; system output can be stored in the variable associated with the return value type. In essence, this means that the data value context can be implemented as a simple map data structure, mapping data types to runtime values.

In case of data dependencies along single data values that are merely passed between method invocations, it is easy to see how this approach can work. However, already the simple e-commerce scenario contains a data dependency that is more sophisticated: the example contains one method that returns a list of products, while another method consumes single product objects. It is obviously not viable to merely pass the complete list of products to the latter method. Instead, a single data value has to be extracted from the overall list of products and subsequently be used as parameter value.

For this reason the data value context of the reusable LearnLib test driver employs a scriptable JavaScript context, as described in Paper V. This context is able not only to retrieve simple named variables (such as the variable “`products`” that stores the complete list of products), but it also can evaluate statements such as “`elementOf(products)`” to isolate single data values using the predefined “`elementOf`” function.

These complex statements are stored as parameters of abstract learning symbols and subsequently evaluated at runtime. They can be generated as part of the type analysis, when sequences of data values are detected as data source for singular parameters.

6.3. The learning setup interchange format by example

The result of the learning setup creation by interface analysis is stored in an interchange format, which subsequently is parsed to produce an actual instance. Figure 6.2 shows an excerpt of a learning setup specification generated for the e-commerce scenario. Following information is specified in this example:

```
 1 <learnsetup>
 2  <serviceurl>http://vulpis.cs.tu-dortmund.de:9000/ecommerceservice?wsdl
       </serviceurl>
 3  <provided>
 4   <object name="username" type="string">username</object>
 5   <object name="password" type="string">password</object>
 6  </provided>
 7  <symbols>
 8   <symbol name="openSession">
 9    <parameters>
10     <parameter>
11      <alternative>username</alternative>
12     </parameter>
13     <parameter>
14      <alternative>password</alternative>
15     </parameter>
16    </parameters>
17    <return>session</return>
18   </symbol>
19   ...
20   <symbol name="getAvailableProducts">
21    <parameters />
22    <return>productArray</return>
23   </symbol>
24   ...
25   <symbol name="addProductToShoppingCart">
26    <parameters>
27     <parameter>
28      <alternative>session</alternative>
29     </parameter>
30     <parameter>
31      <alternative selector="elementOf" field="item">productArray
           </alternative>
32      <alternative selector="elementOf" field="items">shoppingCart
           </alternative>
33     </parameter>
34    </parameters>
35    <return>session</return>
36   </symbol>
37  </symbols>
38 </learnsetup>
```

Figure 6.2.: Excerpt of a learning setup for the e-commerce example. Source: Paper V

- The location of the target system is denoted in line 2.
- An instance pool of predetermined data values is specified in lines 3 to 6. Two variables and initial values are specified (`username` and `password`), which constitute credentials for the authorization step of the target system.
- A description of the alphabet is given, i.e., a list of methods that are to be invoked. In lines 8 to 37 a total of three symbols are defined (`openSession`, `getAvailableProducts`, and `addProductToShoppingCart`) which relate to methods exposed in the SUL's interface.
- For every method the symbolic names of parameters and return values are specified. For instance, the symbol `openSession` has two parameters, receiving values from the variables `username` and `password` respectively. Output will be stored in the variable `session`. The symbol `addProductToShoppingCart` also has two parameters: one parameter reads the `session` variable, while the second parameter expects a single product value. There are two possible sources for product objects, which are referred to in separate <`alternative`> declarations. One source is the list of available products as obtained by the `getAvailableProducts` primitive that was discussed already. Another source (that is not included in the presented configuration excerpt) is the shopping cart, which already may contain items. Both sources offer lists of items, not single values, necessitating the use of the `elementOf` operator. Due to technicalities of the WSDL to Java mapping, this operator has to be applied on a subfield of the respective data structures. This subfield is declared using the `field` attribute. In total, the scripted statement executed by the data value context to retrieve a single data value for the first alternative of the second parameter thus is assembled to `elementOf(productArray.item)`.

Execution of the learning setup shown in Figure 6.2 produces the automata model presented in Figure 6.3. This model contains information that is not deducible from the interface description alone, reflecting actual behavioral and state-dependent traits. For instance, the model shows that shopping carts can only be purchased if at least one product item was added beforehand, and that the shopping cart will be emptied upon purchasing its contents. Note that for the sake of readability only operations that execute successfully are present in Figure 6.3, omitting all “error” transitions.

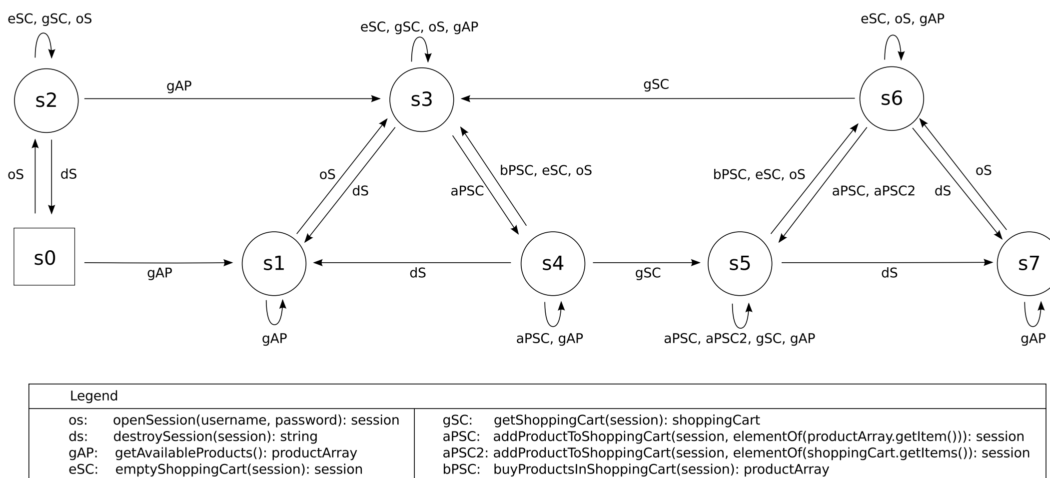


Figure 6.3.: Learned model of the e-commerce example. Source: Paper V

7. Related Work

This chapter explores the context of the work presented in this thesis. Starting with a quick look onto the general field of machine learning, the chapter will then focus on work in the realm of automata learning and conclude with work on tools and real-life applications in this area.

Machine learning The branch of automata learning discussed in this thesis can be regarded to be part of the more general field of *machine learning*. In [51] a definition of machine learning is provided, which essentially says that a computer program can be considered to learn with experience if for a given task and quality measure the results improve with gained experience. It is easy to imagine how this definition applies to the passive automata learning algorithm sketched in Section 2.2, where one expects the accuracy of the predictions made by the created model to improve with the growth of the processed data set, i.e., the processed experience. The same is true for active automata learning, which, however, actively procures the acquisition of experience to satisfy goals of completeness and consistence. Other branches of machine learning include methods such as artificial neural networks [61], bayesian networks [40], decision tree learning [56], and clustering of data sets (e.g., [43]). Common themes of all these approaches include the prediction of as-of-yet unseen data or the classification of new incoming data. In contrast to this, Knowledge Discovery (KD) or Knowledge Discovery in Databases (KDD) [24] tries to mine knowledge in already existing data sets, motivated by the desire to identify relevant information in a "flood of data" without human intervention. Despite differing goals, the automated identification of patterns relates KD/KDD with machine learning.

Automata learning Algorithms for automata learning have been conceived in the 1970ies [11, 64]. These algorithms are concerned with the construction of automata from behavioral traces, i.e., they can be regarded as passive learning algorithms. A complexity result, namely that construction of minimal automata from traces is NP-complete, was provided in [26]. Work on passive learning algorithm continues, e.g., by leveraging existing methods for well studied problems from other domains and reformulating automata construction accordingly. For instance, in [31] the problem of DFA identification is attacked employing SAT solvers.

The concept of employing membership and equivalence queries was first discussed by Angluin in [5], with the L^* algorithm presented in [6]. This algorithm planted the seed for further research in this area, e.g., by optimizing L^* in various ways. Maler and Pnueli [44] introduced handling counterexamples by directly adding suffixes to the observation table instead of the original L^* approach of adding prefixes that lead to inconsistencies which only then are in turned resolved by the introduction of additional suffixes. Rivest and Shapire [60] presented an approach to identify a single suffix for each provided counterexample employing an analysis resembling a binary search, consuming additional membership queries

in the process. Overall, however, this approach usually brings sizable savings on the number of membership queries as the subsequent learning phases consume less queries to form a refined conjecture, compared to setups where all suffixes of a counterexample are considered. Learning algorithms utilizing discrimination trees were introduced by Kearns and Vazirani in [42] and Howar in [32]. For security testing, [29] provides an overview on the current state of the art of regular inference. In that application area, large input sets have to be supported, which is problematic with, e.g., the traditional L^* algorithm due to the generated number of membership queries that grows in a quadratic fashion with the size of the input alphabet. Thus the L_1 algorithm was proposed [37], which avoids initialization of the observation table with one column per input symbol, only adding elements to the set of suffixes that distinguish states.

Whereas the original scope of L^* was limited to learning language acceptors, i.e., DFAs, an extension to Mealy Machines was proposed in [53], in the form of the $L^*_{i/o}$ algorithm. This advancement avoids the powerset-like construction of a learning alphabet composed of pairs of input- and output-symbols that otherwise is necessary when learning models for reactive systems with DFA learning. Further advancing the expressiveness of models that can be obtained via automata learning, Register Automata and an accompanying learning procedure have been proposed [17, 33]. Register Automata allow for expressing the influence of data values on system behavior, which is, e.g., of utmost importance for learning communication protocols. Learning procedures have also been developed for I/O Automata [4], timed Automata [27, 28], Petri Nets [22] and Kripke structures [49].

As the number of queries executed on the SUL has considerable impact on the applicability of learning techniques in real-life examples, various approaches have been developed to reduce the number of queries that have to be processed. Applying application-specific knowledge to filter queries have been discussed in several works [36, 46, 8, 38], as well as the impact of applying several filters in succession [45], where it is demonstrated that the effectiveness of filters is influenced by their order within the processing chain.

Model-based testing methods [16] have been proposed for realizing equivalence queries. If an upper bound for the size of the SUL is established, both the W-method [12] and the Wp-method [25] are guaranteed to find every discrepancy between any considered hypothesis and the target system. Unfortunately, however, these approaches have exponential complexity in the size of the SUL. In practice it is often impossible to determine a specific upper bound, undermining the correctness of these approaches or inflating complexity by resorting to conservative estimates. Instead of proving correctness of a given hypothesis by means of model-based testing, a recent approach is to instead focus on finding counterexamples fast. One such approach that led to winning the ZULU competition [18] is discussed in [34]. The core idea here is to identify parts of the hypothesis that are either guaranteed to be correct (i.e., the spanning tree of states explored by the learning procedure) or well-tested (parts that already were tested in the accumulated runs of equivalence queries) and to focus testing on other parts.

As discussed in Section 6.2.3, the management of concrete data values during interaction with real-life systems is an important enabler for learning real-life systems. In the proposed solution, this is achieved during steps of abstraction and concretization, employing a fixed mapping determined by interface analysis between the realm of the abstract and the concrete. Another approach is Automatic Abstraction Refinement [35], where new abstract

learning symbols with an accompanying concrete data value will be introduced whenever counterexamples reveal that a certain concrete data value reveals behavior that is not covered by an existing abstract symbol and its related concretization. The topic of finding fitting abstractions for automata learning is also discussed in [2] and [21].

Tools and applications Aside of the various iterations of LearnLib, other implementations of active automata learning have been produced as well. Most notable is libalf [14], which is an open-source library with learning algorithms which are implemented in C++. This library offers extensive support for observing data structures within learning algorithms. Compared to LearnLib, however, it does not have an equally pronounced focus on real-life learning applications, shipping with little accompanying infrastructure.

LearnLib in its initial incarnation was first presented in [59], receiving additional coverage in [47]. Much like libalf nowadays, this version of LearnLib was implemented in C++. In contrast to the current Java version of LearnLib, no unifying component model was present.

Automata learning has seen numerous applications from various domains. In [30], learning is used to generate models of CTI (Computer Telephony Integration) systems, marking the advent of practical learning scenarios for real-life systems. In [57, 58] record and replay testing is combined with active automata learning (provided by LearnLib) in the context of web applications. It is shown that models generated in this fashion can be used, e.g., to ensure security properties (such as that restricted areas of the web application can only be accessed after a system login) by means of model checking. Models for the backend of an enterprise web application were learned in [8] and [9]. In [3] learning techniques were employed to learn models for a biometric passport system, revealing spurious behavior outside of the system's specification that can most likely be attributed to problems with the employed system interface. The application of learning techniques on communication protocol entities is illustrated in [13]. In the realm of security research, automata learning has been employed to learn models of "botnets", i.e., networked clusters of systems infected with malicious software that can, for example, be used by criminals to conduct cybercrimes. Results in that area are presented in [15]. Automata learning was also applied onto automotive embedded systems [62]. This work includes an estimate and breakdown for the effort spent on the overall procedure, showing that manual construction of learning setups can take a considerable portion of the overall effort. Another application area recently explored are wireless sensor networks, discussed in [21], where learning is utilized to create interface automata [19] models.

8. Conclusion and Outlook

8.1. Conclusion

This thesis discussed the development of an infrastructure for active automata learning that can drive the adoption of active automata learning techniques for real-life applications. This necessitated work in several areas:

Framework and Tools: The LearnLib library for active automata learning was completely restructured, replacing a C++ implementation without clear modularization with a Java framework that is structured by interface declarations reflecting and extending the architecture of the MAT learning paradigm. Implementations of learning algorithms and supporting infrastructure were developed within this framework and provide a flexible and easy to use starting point for the development of learning setups.

To enable easy access to LearnLib components, a graphical solution for modeling learning setups was implemented on the basis of the jABC framework. The resulting LearnLib Studio inherits jABC features such as execution of modeled graphs and model checking. Being based on the concept of service oriented orchestration, modeled learning solutions can also be embedded within management processes, e.g., to facilitate repeated learning for regression testing.

Algorithms: The new DHC algorithm is inspired by the well-known breadth-first search, constructing a hypothesis along the course of the SUL exploration. In contrast to classical learning algorithms employing an observation table as central data structure, the DHC algorithm maintains a hypothesis at all times. Thus construction progress can be made visible at any time, making DHC unusually easy to follow and establishing this algorithm as a good choice for didactic purposes.

The unique construction approach enables the implementation of the DHC algorithm with data structures that have clearly separated purposes:

- The hypothesis reflects the construction process and is, once completed, the primary artifact.
- The work list drives the future exploration.
- A data structure associating output behavior to states is used to determine candidates for state equivalence.
- A query cache removes redundant queries that occur during hypothesis refinement. It can also serve as a prefix filter.

This clear separation of concern enables efficient implementation of externalized versions of these data structures, optimized for their respective typical access patterns. With the ability to choose between internal or external data structures, the memory profile of the DHC algorithm can be tailored for the application area at hand.

Enabling scalability: Active automata learning of real-life systems can bind considerable resources in terms of time, computation and storage. The LearnLib framework offers means to overcome these problems, e.g., by allowing for the integration of application-fit filters that can dramatically reduce the number of queries that have to be executed on the target system. Beyond that, the support for query batches allows for parallelized execution of queries on duplicated SUL instances. Experimental results confirm that significant time savings can be achieved by processing learning queries in a networked cluster.

The instantiation of several target system instances, however, requires the allocation of additional resources. Thus it makes sense to consider the use of cloud computing to allocate resources on demand instead of costly (over-) provisioning of local resources. The WebABC prototype demonstrates that completely cloud-based solutions for modeling and executing learning setups can be realized.

Automated configuration of learning setups: Even with the development of efficient active automate learning technology and accompanying tools, one major issue that hinders adoption of learning methods into real-life practice is the setup effort needed to conduct application-fit learning experiments. This can be mostly attributed to the need of constructing fitting alphabets and test drivers that can execute actions on the SUL accordingly.

This effort can be dramatically reduced by employing a reusable test driver that can be configured for the target system at hand. Configurations can be generated by means of interface analysis, including strategies for dealing with runtime data.

8.2. Outlook

The work presented in this thesis can be extended in a lot of ways. No framework is "complete" and it is hard to imagine that algorithms for automata learning do not have room left for improvement.

LearnLib and LearnLib Studio: The LearnLib framework implements learning according to the (extended) MAT model. However, active automata learning is not always applicable (e.g., if the target system is in productive mode and cannot be duplicated), so including passive learning algorithms may be a worthwhile undertaking to harvest automata models, e.g., from log files.

Another area where additional work can be done is the implementation of additional automata models and corresponding learning algorithms.

LearnLib Studio can be extended to include, for instance, wizard-style configuration facilities that automatically propose learning setups according to the SUL's interface specification. Including technology to automatically configure learning setups presented in this thesis, the overall effort to create learning setups can be significantly reduced.

DHC algorithm: The current DHC algorithm resumes construction from-scratch when refining the hypothesis. This allows for an easy implementation of the algorithm, without special cases beyond adding additional symbols to the learning alphabet when evaluating counterexamples. However, it should be possible to avoid rebuilding the complete hypothesis as all discovered states are guaranteed to be distinct (and thus will also be in the final result) and all transitions that were not created during state merging are part of the spanning tree of the automaton and also guaranteed to be correct. By exploiting these properties a scheme to locally repair an inaccurate hypothesis should be feasible that avoids complete reconstruction.

Cloud learning: This thesis presents initial work to move learning into a cloud environment. While this approach looks promising, the overall applicability still has to be demonstrated with a series of case studies. Challenges remain in the area of actual tool maturity, security and reliability.

Bibliography

[1] Fides Aarts, Johan Blom, Therese Bohlin, Yu-Fang Chen, Falk Howar, Bengt Jonsson, Maik Merten, Ralf Nagel, Antonino Sabetta, Siavash Soleimanifard, Bernhard Steffen, Johan Uijen, Thomas Wilk, and Stephan Windmüller. Establishing basis for learning algorithms, technical report. `http://hal.inria.fr/inria-00464671/PDF/connect_WP4_D41.pdf`, 2010.

[2] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[3] Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and Abstraction of the Biometric Passport. In Margaria and Steffen [48], pages 673–686.

[4] Fides Aarts and Frits Vaandrager. Learning I/O Automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin / Heidelberg, 2010.

[5] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981.

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

[7] Marco Bakera, Tiziana Margaria, Clemens Renner, and Bernhard Steffen. Tool-supported enhancement of diagnosis in model-driven verification. *Innovations in Systems and Software Engineering*, 5:211–228, 2009. 10.1007/s11334-009-0091-6.

[8] 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.

[9] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing System States by Active Learning Algorithms. In Alessandro Moschitti and Riccardo Scandariato, editors, *Eternal Systems*, volume 255 of *Communications in Computer and Information Science*, pages 61–78, Budapest, Hungary, 2012. Springer Verlag.

[10] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[11] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[12] Avrim Blum and Steven Rudich. Fast learning of k-term DNF formulas with queries. In *Proc.* 24th *ACM Symp. on Theory of Computing*, pages 382–389, New York, NY, USA, 1992. ACM.

[13] Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In Margaria and Steffen [48], pages 658–672.

[14] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The Automata Learning Framework. In *CAV'10*, pages 360–364, 2010.

[15] Georges Bossert, Guillaume Hiet, and Thibaut Henin. Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems. In *Proceedings of the 2011 Conference on Network and Information Systems Security*, pages 1–8, La Rochelle, France, June 2011. IEEE.

[16] 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 New York, Inc., Secaucus, NJ, USA, 2005.

[17] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A succinct canonical register automaton model. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer Berlin Heidelberg, 2011.

[18] David Combe, Colin de la Higuera, Jean-Christophe Janodet, and Myrtille Ponge. Zulu - Active learning from queries competition. `http://labh-curien.univ-st-etienne.fr/zulu/index.php`. Version from 01.08.2010.

[19] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[20] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[21] Faranak Heidarian Dehkordi. Studies on verification of wireless sensor networks and abstraction learning for system inference. Doctoral thesis, Radboud Universiteit Nijmegen, 2012.

[22] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning Workflow Petri Nets. In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 206–225. Springer Berlin / Heidelberg, 2010.

[23] Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema. `http://www.w3.org/TR/sawsdl/`, 2007. Version from 05.09.2012.

[24] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57–70, 1992.

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

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

[27] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. In *In 6th International Workshop on Verification of Infinite-State Systems, volume 138/4 of Electronic Notes in Theoretical Computer Science*, pages 379–395. Springer, 2004.

[28] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In *Proceedings of the 17th international conference on Concurrency Theory*, CONCUR'06, pages 435–449. Springer, 2006.

[29] Roland Groz, Muhammad-Naeem Irfan, and Catherine Oriat. Algorithmic improvements on regular inference of software models and perspectives for security testing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 444–457. Springer Berlin / Heidelberg, 2012.

[30] Andreas Hagerer, Hardi Hungar, Tiziana Margaria, Oliver Niese, Bernhard Steffen, and Hans-Dieter Ide. Demonstration of an operational procedure for the model-based testing of cti systems. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 336–340. Springer, 2002.

[31] Marijn J. H. Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *Proceedings of the 10th international colloquium conference on Grammatical inference: theoretical results and applications*, ICGI'10, pages 66–79, Berlin, Heidelberg, 2010. Springer-Verlag.

[32] Falk Howar. Active learning of interface programs. Doctoral thesis, TU Dortmund, Department of Computer Science, Chair of Programming Systems, 2012.

[33] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.

[34] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In Margaria and Steffen [48], pages 687–704.

[35] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin / Heidelberg, 2011.

[36] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-Specific Optimization in Automata Learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. 15th Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, July 2003.

[37] Muhammad Naeem Irfan, Roland Groz, and Catherine Oriat. Improving model inference of black box components having large input test set. In *Proceedings of the 11th International Conference on Grammatical Inference, ICGI 2012*, pages 133–138, September 2012.

[38] 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.

[39] 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 *14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 154–161. IEEE, 2009.

[40] F. Jensen. *An Introduction to Bayesian Networks*. Springer Verlag, New York, 1996.

[41] Bengt Jonsson. Learning of automata models extended with data. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer, 2011.

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

[43] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

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

[45] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Analyzing second-order effects between optimizations for system-level test-based model generation. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 7 pp. –467. IEEE, 2005.

[46] 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.

[47] Tiziana Margaria, Harald Raffelt, Bernhard Steffen, and Martin Leucker. The LearnLib in FMICS-jETI. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 340–352, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*. Springer, 2010.

[49] Karl Meinke and Muddassar Sindhu. Incremental Learning based Testing for Reactive Systems. In *Proceedings of the 5th international conference on Tests and proofs*, number 6706 in Lecture Notes In Computer Science, pages 134–151. Springer, 2011.

[50] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, 2011. Version from 02.09.2012.

[51] T. Mitchell. *Machine Learning (Mcgraw-Hill International Edit)*. McGraw-Hill Education (ISE Editions), 1st edition, October 1997.

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

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

[54] Object Management Group (OMG). CORBA website. `http://www.corba.org/`. Version from 09.09.2012.

[55] Oracle Corporation. Trail: The Reflection API. `http://docs.oracle.com/javase/tutorial/reflect/index.html`. Version from 29.10.2012.

[56] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[57] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, New York, NY, USA, 2008. ACM.

[58] 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.

[59] Harald Raffelt and Bernhard Steffen. Learnlib: A library for automata learning and experimentation. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer, 2006.

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

[61] David E. Rumelhart, Bernard Widrow, and Michael A. Lehr. The basic ideas in neural networks. *Commun. ACM*, 37(3):87–92, March 1994.

[62] Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative refinement of specification for component based embedded systems. In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 276–286. ACM, 2011.

[63] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-Driven Development with the jABC. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Hardware and Software, Verification and Testing*, volume 4383 of *Lecture Notes in Computer Science*, pages 92–108. Springer Berlin / Heidelberg, 2007.

[64] B.A. Trakhtenbrot. *Finite automata; behavior and synthesis*. Fundamental studies in computer science. North-Holland Pub. Co., 1973.

[65] Ben Treynor. Gmail back soon for everyone. `http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html`, 2011. Version from 02.09.2012.

[66] World Wide Web Consortium (W3C). Web services. `http://www.w3.org/2002/ws/`. Version from 09.09.2012.

A. Selected 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 **Demonstrating Learning of Register Automata**
by Maik Merten, Falk Howar, Bernhard Steffen, Sofia Cassel, and Bengt Jonsson. In Cormac Flanagan and Barbara König, Barbara, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7214:466-471.

III **Automata Learning with on-the-Fly Direct Hypothesis Construction**
by Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. In Reiner Hähnle et al., editors, *ISoLA 2011 Workshops*, 2012, *Communications in Computer and Information Science*, *Springer Verlag*, 336:248-260.

IV **The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning**
by Falk Howar, Oliver Bauer, Maik Merten, Bernhard Steffen, and Tiziana Margaria. In Reiner Hähnle et al., editors, *ISoLA 2011 Workshops*, 2012, *Communications in Computer and Information Science*, *Springer Verlag*, 336:232-247.

V **Automated Learning Setups in Automata Learning**
by Maik Merten, Malte Isberner, Falk Howar, Bernhard Steffen, and Tiziana Margaria. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 5th International Symposium on Leveraging Applications, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7609:591-607.

VI **Dynamic Testing via Automata Learning**
by Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. In *International Journal on Software Tools for Technology Transfer, Volume 11*, 2009, *Springer Verlag*, 307-324.

B. Comments on my participation

- **Paper I**
  The concept and the contents of this paper were discussed amongst all authors. I designed and described the running example. I am main author of the parts that describe the DHC algorithm and am coauthor of the other sections.

- **Paper II**
  I am one of the two main developers of the described software and implemented the presented modeling approach for learning setups. The design of the component model that allowed for the rapid integration of Register Automata was authored by me. This design is a major contribution and formed the basis for subsequent innovation.

- **Paper III**
  I am the main author of this paper and invented the presented algorithm, which is a core contribution also presented in this thesis. The algorithm and its presentation was discussed amongst all authors.

- **Paper IV**
  The concept of generating multiple learning queries at once for parallelized processing was discussed and developed by the authors of this paper over several months. I integrated this concept into the underlying software framework and implemented core components for the experimental evaluation.

- **Paper V**
  I am the main author of the paper. The reconfigurable test driver and the setup interchange format were created by me, enabling highly automated learning setups. I designed and implemented the target system used as a running example.

- **Paper VI**
  The concept of learning web applications in the described fashion was developed and discussed amongst all authors. I implemented parts of the test execution logic and the test result aggregator.

C. Other publications

- **Automated Inference of Models for Black Box Systems based on Interface Descriptions**
  by Maik Merten, Falk Howar, Bernhard Steffen, Patricio Pellicione, and Massimo Tivoli. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 5th International Symposium on Leveraging Applications, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7609:79-96.

- **LearnLib Tutorial: From Finite Automata to Register Interface Programs**
  by Falk Howar, Malte Isberner, Maik Merten, and Bernhard Steffen. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 5th International Symposium on Leveraging Applications, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7609:587-590.

- **The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems**
  by Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 5th International Symposium on Leveraging Applications, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, 2012, *Lecture Notes in Computer Science*, *Springer Verlag*, 7609:608-614.

- **Simplicity driven application development**
  by Maik Merten and Bernhard Steffen. *presented at Society for Design and Process Science, SDPS 2012, Berlin*

- **Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring**
  by Antonia Bertolino, Antonello Calabrò, Maik Merten, and Bernhard Steffen. In *ERCIM News, Volume 88*, 2012

- **A Succinct Canonical Register Automaton Model**
  by Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6996:366-380.

- **Next Generation LearnLib**
  by Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6605:220-223.

- **Automata Learning with Automated Alphabet Abstraction Refinement**
  by Falk Howar, Bernhard Steffen, and Maik Merten. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, 2011, *Lecture Notes in Computer Science*, *Springer Verlag*, 6538:263-277.

- **From ZULU to RERS - Lessons Learned in the ZULU Challenge**
  by Falk Howar, Bernhard Steffen, and Maik Merten. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, 2010, *Lecture Notes in Computer Science*, *Springer Verlag*, 6415:687-704.

- **On Handling Data in Automata Learning - Considerations from the CONNECT Perspective**
  by Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II*, 2010, *Lecture Notes in Computer Science*, *Springer Verlag*, 6416:221-235.

- **Hybrid Test of Web Applications with Webtest**
  by Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. In Tevfik Bultan and Tao Xie, editors, *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), TAV-WEB 2008, Seattle, Washington, USA, July 21, 2008*, 2008, *TAV-WEB*, *ACM*, 1-7.