State Machine Learning Using Queries

Piet van Agtmaal Stefan Boodt Sander Bosma Gerlof Fokkema
Jente Hidskes Arjan Langerak Skip Lentz

Abstract

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Introduction

Active state machine learning is a framework in which a learning algorithm (the learner) is given access to a teacher to ask questions of different kind. The goal of active state machine learning is to identify an unknown state machine. It was first introduced by Dana Angluin in her paper *Learning Regular Sets from Queries and Counterexamples* in 1987[6].

Active state machine learning tools are becoming increasingly prevalent in today's world for many different applications.

add references?

Examples of such applications are protocol implementation validation and detection of botnets.

add references, more examples? (E.g. software testing)

There are a plethora of papers and articles written about active state machine learning. Many, however, use different terms, slight variations on algorithms or data structures or are otherwise difficult to read. This leads to the situation in which it is hard to get a proper overview of the subject.

The goal of this article, therefore, is to give a global overview of where the technology of active state machine learning is currently at. While doing so, an attempt is made to bring together all different terms used and to explain the slight variations in algorithms. For this reason, this article should serve the purpose of giving the reader a quick, concise but complete overview of not only the current state of the art, but also the history of active state machine learning and important advances made within this field of study over time.

To accomplish this, the first chapter explains the fundamental theory behind the L^* algorithm for learning regular sets. Chapter two discusses improvements to the L^* algorithm. Variants of active state machine learning are highlighted in chapter three.

The remaining chapters are concerned with applications of active state machine learning. Chapter four discusses the gap that has to be bridged between the concepts used in the L^* algorithm and the real life problem domain(s). Chapter five discusses several tools that have been created to apply active state machine learning. Lastly, important real world applications are highlighted in chapter six.

1 Fundamental Theory

This part introduces the theory fundamental to active state machine learning: the notion of Nerode-equivalence, and the original learning algorithm due to Angluin called L^*. Before this, an explanation of the notation used throughout the article is in order.

1.1 Basic Notation

Define Σ^* to be all of the words constructed from the input alphabet Σ. Furthermore, ϵ represents the empty word. Note that by definition it holds that $\epsilon \in \Sigma^*$. The concatenation operator is denoted $v \cdot w$ for two strings $v, w \in \Sigma^*$, or for elements in the input alphabet Σ. When $\cdot$ is applied to sets, it equals the concatenation of all the elements in the Cartesian product of the two sets.

Definition 1.1 below shows the definition of a DFA used in the article.

Definition 1.1 (Deterministic Finite Automaton) *A DFA M is a 5-tuple $(Q, q_0, \Sigma, \delta, F)$ where Q is the set of states, q_0 the initial state, Σ the input alphabet, δ the transition function, and F the set of accepting states.*

For some word $w \in \Sigma^*$, let $\delta^*(q, w)$ denote the state reached by processing w from state q. Furthermore, denote $M[w] = \delta^*(q_0, w)$ to be the state reached by processing w from the initial state q_0, as used by Kearns and Vazirani[**?**]. Then w is called the *access string* of the state $M[w]$ (see definition 1.2). Lastly, let $\mathcal{L}_M$ denote the regular language accepted by M; that is $\mathcal{L}_M = \{w \mid M[w] \in F\}$.

Definition 1.2 (Access String) *For some alphabet Σ and a state $q \in Q$ of some DFA M, a string $w \in \Sigma^*$ is called an* access string *of q if $M[w] = q$*

1.2 Nerode-Equivalence and Equivalence Classes

It is possible for a DFA M to contain states which are redundant. That is, for some states, any word processed from all of those states leads to an accepting state or a rejecting state. More formally, for two states $p, q \in Q$ and a word $w \in \Sigma^*$, both $\delta^*(p, w)$ and $\delta^*(q, w)$ are either an accepting state or a rejecting state. There is no w which *distinguishes* the two states.

From this point of view, the states p and q can be considered equivalent, in that they pose no difference for any word processed from those states to be accepted or not. The access strings v and w of p and q, respectively, are said to be *Nerode-equivalent*, as described in definition 1.4. A group of access strings form an *equivalence class* if they are mutually Nerode-equivalent. Thus any set of access strings can be partitioned into equivalence classes.

Definition 1.3 (Distinguishing Extension) *Let $\mathcal{L}$ be a language over some alphabet Σ, and let v, w and z be words in Σ^*. Then z is a* distinguishing extension *for v and w if and only if exactly one of $v \cdot z$ and $w \cdot z$ is in $\mathcal{L}$.*

Definition 1.4 (Nerode-equivalence) *For some language $\mathcal{L}$ over an alphabet Σ, define two words $v, w \in \Sigma^*$ to be* Nerode-equivalent *with respect to $\mathcal{L}$ if and only if there is no distinguishing extension z for v and w. We write $v \equiv_{\mathcal{L}} w$ if v and w are Nerode-equivalent.*

Let $[w]_{\mathcal{L}}$ denote the equivalence class *of an access string w, such that for any word v in $[w]_{\mathcal{L}}$ it holds that $v \equiv_{\mathcal{L}} w$. Lastly, let $\equiv_{\mathcal{L}}$ be the set of all equivalence classes of $\mathcal{L}$. For convenience, denote $\equiv_M$ to be the same set for some DFA M.*

From the above definition 1.4, the Myhill-Nerode theorem makes a strong statement: a language $\mathcal{L}$ is regular if and only if the set of equivalence classes $\equiv_{\mathcal{L}}$ is a finite set. Indeed, this theorem is stronger than the pumping lemma for regular languages, which can only prove that a language is *not* regular.

According to Hopcroft and Ullman, it follows from the Myhill-Nerode theorem that each regular language $\mathcal{L}$ has a minimal DFA M which accepts $\mathcal{L}$. In fact, the amount of states of M is precisely equal to the amount of equivalence classes in $\equiv_{\mathcal{L}}$, thus there is a one-to-one correspondence between $\equiv_{\mathcal{L}}$ and the states of M.

Thus, from a set of equivalence classes $\equiv_M$, M can be constructed as follows. Let $\equiv_M$ represent the set of states of M. For each state $[w]_{\mathcal{L}_M}$ and for each $\sigma \in \Sigma$, an outgoing transition can be constructed to the state $[w \cdot \sigma]_{\mathcal{L}_M}$. The initial state would be represented by the equivalence class $[\epsilon]_{\mathcal{L}_M}$. Lastly, a state $[w]_{\mathcal{L}_M}$ is an accepting state if and only if $[w]_{\mathcal{L}_M} \subseteq \mathcal{L}_M$.

1.3 Key Principle of Active Learning

In active state machine learning, the goal is to learn an unknown finite state automaton M (the *target*). The key principle of all active state machine learning algorithms is to progressively refine a hypothesis automaton $\hat{M}$ until it equals M.

Put in terms of equivalence classes, a learning algorithm progressively refines a set of equivalence classes $\equiv_{\hat{M}}$, until it eventually converges to the set of equivalence classes $\equiv_M$ of the target.

To ensure progress at each step, any active learning algorithm has access to a so-called *minimally adequate teacher* (MAT) which can answer two types of queries concerning the target. This is what sets *active* state machine learning apart from *passive* state machine learning, where there is no notion of a MAT or of queries. The two types of queries are the following:

Membership Query Given a word, answers "yes" or "no" depending on whether it is accepted by the target M or not.

Equivalence Query Given a hypothesis DFA $\hat{M}$, answers "yes" if $\hat{M}$ equals M, in which case the algorithm is finished. If the hypothesis is not equal, it provides a counterexample γ. The counterexample can be used to further refine $\equiv_{\hat{M}}$, since γ is an example of an incorrect equivalence class.

Note that for the equivalence query, the hypothesis can be constructed in a manner similar to the construction of a minimal DFA described in the previous section.

If a learning algorithm ensures progress at each step using the above queries, the algorithm is guaranteed to converge to a target automaton M. This holds because $\equiv_M$ must be finite for some DFA M according to the Myhill-Nerode theorem, thus the amount of progress that can be made must also be finite. The same can be said for other types of automata for which the Myhill-Nerode theorem is applicable. An example is the Mealy machine, which will be the scope of discussion of section 3.

Before that, however, the original algorithm due to Angluin for learning DFAs will be discussed.

1.4 An Algorithm for Learning DFAs

In 1987, Angluin published the L^* algorithm for learning DFAs[?]. Since all of the work on active state machine learning is based on the work due to Angluin, a section is devoted to describe the algorithm in detail.

The first part explains the data structure which maintains the necessary information for constructing a hypothesis. After that, the algorithm is described. To get a more intuitive sense of the workings of the algorithm, an example demonstration of the algorithm is given. The section then concludes with a complexity analysis.

1.4.1 The Data Structure

While learning, the algorithm maintains a set of access strings S (see definition 1.2) and a set of distinguishing extensions E (see definition 1.3) in a two-dimensional table called the *observation table*. The columns of this table are labelled by the items of E. The rows are split into an upper part and a lower part, which are labelled by items from S and $S \cdot \Sigma$ respectively. Lastly, the cells corresponding with a row s and a column e, are labelled with 1 or 0 depending on whether $s \cdot e \in \mathcal{L}_M$. The observation table is initialized with $S = E = \{\epsilon\}$, as can be seen in table 1a.

An observation table provides sufficient information for constructing a hypothesis, as each row is a representation of an equivalence class of $\mathcal{L}_M$. Let $\mathrm{row}(s)$ represent the finite function $f : E \to \mathbb{B}$ such that $f(e) = (s \cdot e \in \mathcal{L}_M)$. Then $\mathrm{row}(s)$ has a direct correspondence to the equivalence class $[s]_{\mathcal{L}_M}$. Moreover, the rows in the lower part of the observation table represents the equivalence classes needed for constructing the state transitions. Lastly, since the table is initialized with ϵ, the initial state can also be constructed. Thus a hypothesis automaton $\hat{M}$ can be constructed following the same procedure described in section 1.2.

However, to ensure that the constructed transitions are correct, the algorithm has to maintain two invariants related to the observation table: one of closure and of consistence, defined in definitions 1.5 and 1.6 respectively.

Definition 1.5 (Closure) *Let $t \in S \cdot \Sigma$. An observation table is closed if there exists some $s \in S$ such that* $\mathrm{row}(t) = \mathrm{row}(s)$.

Definition 1.6 (Consistence) *Let $s_1, s_2 \in S$ be two strings such that* $\mathrm{row}(s_1) = \mathrm{row}(s_2)$. *An observation table is consistent if for all $\sigma \in \Sigma$ it holds that* $\mathrm{row}(s_1 \cdot \sigma) = \mathrm{row}(s_2 \cdot \sigma)$.

1.4.2 The Algorithm

As said in the previous section, the algorithm begins by initializing S and E to $\{\epsilon\}$. The initial cells of the table are filled by performing membership queries. After the initialization is done, the algorithm enters the main loop, which is split up into two phases. The first phase ensures the properties of closure and consistence, and is repeated until these properties are satisfied. In the second phase the algorithm constructs a hypothesis automaton, and uses it to perform an equivalence query.

The first phase If the property of closure (definition 1.5) is not satisfied, then there exists some $t \in S \cdot \Sigma$ such that $\mathrm{row}(t) \neq \mathrm{row}(s)$ for all $s \in S$. This means that when constructing a hypothesis $\hat{M}$, there exists no state $\mathrm{row}(t)$ to which a transition can be made. To address this problem, t is added to S and the table is filled by executing the required membership queries.

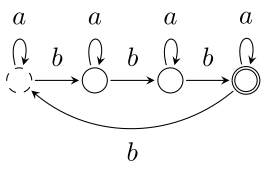
Furthermore, if the property of consistence (definition 1.6) is not satisfied, then there exists some $s_1, s_2 \in S$, $e \in E$ and $\sigma \in \Sigma$ such that $\mathrm{row}(s_1) = \mathrm{row}(s_2)$, but the value in the cell corresponding with $s_1 \cdot \sigma \cdot e$ is unequal with the one corresponding with $s_2 \cdot \sigma \cdot e$. Thus, $\sigma \cdot e$ defines a new distinguishing extension, and is added to E. Again, the table is filled by executing membership queries where needed.

The second phase By now the observation table is both closed and consistent. The algorithm uses the observation table to construct a hypothesis in the manner defined in section 1.2. Once the hypothesis is constructed, the equivalence query is executed upon it. If the response is "yes", the algorithm is done and halts. If instead the response is a counterexample, the counterexample and all of its prefixes are added to S, after which $S \cdot \Sigma$ is updated and the algorithm moves back to phase 1.

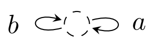
1.4.3 Example run

Suppose L^* is used to learn the target DFA M of figure 1a. The input alphabet for this DFA is $\Sigma = \{a, b\}$. L^* initializes S and E to $\{\epsilon\}$, and performs three membership queries to build the initial observation table of table 1a.

Since the table is both closed and consistent, the algorithm moves on to phase 2 and constructs the hypothesis of figure 1b. It executes an equivalence query with the hypothesis as parameter. Since it does not match M, the oracle will provide a counterexample. Assume the counterexample provided is *bbb* (it could also have provided other counterexamples such as *bbbbbbb*). Now, the



(a) The target



(b) First hypothesis

Figure 1: Automata

	ϵ
ϵ	0
a	0
b	0

(a)

	ϵ
ϵ	0
b	0
bb	0
bbb	1
a	0
ba	0
bba	0
$bbba$	1
$bbbb$	0

(b)

	ϵ	b
ϵ	0	0
b	0	0
bb	0	1
bbb	1	0
a	0	0
ba	0	0
bba	0	1
$bbba$	1	0
$bbbb$	0	0

(c)

	ϵ	b	bb
ϵ	0	0	0
b	0	0	1
bb	0	1	0
bbb	1	0	0
a	0	0	0
ba	0	0	1
bba	0	1	0
$bbba$	1	0	0
$bbbb$	0	0	0

(d)

Table 1: Observation tables

counterexample and all prefixes are added to S. After performing the new membership queries, the table 1b is built and the algorithm returns to phase 1.

The new table is closed, but not consistent, since $\text{row}(b) = \text{row}(bb)$, but $\text{row}(b \cdot b)$ and $\text{row}(bb \cdot b)$ differ under the column ϵ. Therefore, L^* adds $b \cdot \epsilon = b$ to E, resulting in table 1c. This table is still inconsistent, since $\text{row}(\epsilon) = \text{row}(b)$, but $\text{row}(\epsilon \cdot b)$ differs from $\text{row}(b \cdot b)$ under the column b. Therefore, L^* adds $b \cdot b$ to E. Performing the new membership queries now results in table 1d. The table is now both consistent and closed, so L^* moves on to phase 2 and constructs the hypothesis of figure 1a. Since this is equal to the target, the equivalence query returns 'yes' and the algorithm terminates.

1.5 Complexity

Since the runtime complexity is dependent on the implementation of the queries (which are application specific), it is more useful to consider the number of queries made. Specifically, the number of membership queries, as these normally far outnumber the equivalence queries. The asymptotic number of membership queries is called the *query complexity.*

Angluin states that the number of items in the observation table is at most

$$(k + 1)(n + m(n - 1))n = \mathcal{O}(mn^2) \tag{1}$$

In this equation $k = |\Sigma|$, n is the number of states in the minimum acceptor of the target and m is equal to length of the longest counterexample[6]. Since each item has a length of at most $\mathcal{O}(m+n)$, she concludes that the total space required for storing the observation table is $\mathcal{O}(m^2n^2 + mn^3)$.

In section 2 various improvements are discussed. The complexities of these improvements usually contain a term k. In order to be able to directly compare these improvements to L^*, we need to slightly modify Angluin's result. By keeping k as a variable instead of a constant, the upperbound of the number of items in the observation table is $\mathcal{O}(kmn^2)$ (this follows directly from the left-hand side of equation 1). We conclude that the query complexity is also $\mathcal{O}(kmn^2)$, since for each item in the observation table exactly one membership query is performed.[1] The total space required for the observation table becomes $\mathcal{O}(kmn^2) \cdot \mathcal{O}(m+n) = O(km^2n^2 + kmn^3)$. This is equal to the total space complexity, since adding the space required for the hypothesis (which is $\mathcal{O}(kn)$) does not change the complexity.

2 Improvements

Since Angluin's original paper, several improvements have been made with respect to the original L^* algorithm described in section 1. Some of these will be discussed in this chapter. In discussing them, we make two kinds of comparisons between the algorithms.

The first comparison is in terms of the amount of *queries* (the *query complexity*), which gives an indication of the running time of the algorithm. The second comparison is in terms of the *space complexity*. We also discuss the TTT-algorithm, which according to Isberner et al. exhibits an optimal space complexity[24].

The first improvement that will be discussed is *classification trees*, which forms the basis for multiple subsequent improvements (including the TTT-algorithm).

2.1 Classification Trees

Introduced by Kearns and Vazirani[28], classification trees (or *discrimination trees*) are meant as a replacement for the observation table (see section 1.4.1). In other words, it is a different data structure for storing the sets S and E of access strings and distinguishing extensions. The main characteristic of a classification tree is that it is *redundancy-free*, which will be explained in this section.

Say that we have a classification tree for some target automaton M. The labels of the internal nodes of the tree are distinguishing extensions from the set E, and the labels of leaf nodes are access strings from S. The tree's structure is such that for any internal node $e \in E$, its left subtree contains all $s \in S$ such that $s \cdot e$ is rejected by M, and its right subtree contains those s such that $s \cdot e$ is accepted by M.

From the way the tree is constructed, any pair of access strings $v, w \in S$ are distinguished by their *lowest common ancestor* in the tree. Thus, v and w are distinguished by exactly *one* distinguishing extension. In other words, any pair of access strings in the classification tree are Nerode-inequivalent (see definition 1.4), and thus each access string uniquely identifies a distinct equivalence class of the target automaton M. When we construct a hypothesis, each equivalence class (and thus each access string), corresponds to a state in our hypothesis automaton $\hat{M}$.

Howar et al. state that a discrimination tree is *redundancy-free* because each state in $\hat{M}$ is distinguished by one distinguishing extension, as opposed to observation tables where states are distinguished by a fixed amount of distinguishing extensions[16].

2.1.1 Complexity analysis

With their introduction of the *classification tree*, Kearns and Vazirani also adapted the original L^* learning algorithm to use the tree structure[28]. The precise operation of the adapted algorithm will not be given, but we do give an overview of the impact on the query complexity and the space complexity.

Query Complexity The amount of *membership queries* for hypothesis construction is bounded by $\mathcal{O}(kn^2)$ membership queries in the worst-case (i.e. a degenerated tree)[16, 28, 24], where $k = |\Sigma|$ and n is the size of the target automaton M. For the counterexample analysis the amount of queries is bounded by $\mathcal{O}(nm)$, where m is the size of the longest counterexample[28]. Thus in total the query complexity is bounded by $\mathcal{O}(kn^2 + nm)$, an improvement over that of L^*.

[1] Angluin notes that the maximum number of equivalence queries is n, thus for L^* the membership queries indeed far outnumber the equivalence queries.

Space Complexity In chapter 4 of his dissertation, Isberner analyzed the space complexity of discrimination-tree based learners as follows: a degenerated tree has $\mathcal{O}(n)$ amount of nodes, of which each inner node is labeled by a distinguishing extension of length $\mathcal{O}(m)$[23]. The space required for the hypothesis automaton is $\mathcal{O}(kn)$[23]. Thus the algorithm requires space in $\mathcal{O}(kn + mn)$[24, 23]

Compare with L-star

.

2.2 Rivest and Schapire's Counterexample Analysis

In 1993, Rivest and Schapire published a paper on learning without a reset using homing sequences[35], which will be the scope of discussion of section 3.4.

Their paper, however, also included an improvement upon L^*, also in the case that a reset *is* available. Specifically, they improved the *counterexample analysis* stage, which is the part that handles counterexamples returned by an equivalence query.

When a counterexample γ is obtained, Rivest and Schapire's improved algorithm finds a *single* extension e from γ distinguishing two states in the new hypothesis, and adds it to the set E of distinguishing extensions. This is in contrast to the original L^* algorithm, which adds all prefixes of the counterexample to the set S of access strings.

According to Rivest and Schapire[35], and further clarified by Steffen[38] and Isberner[26], there must be a point where the hypothesis automaton $\hat{M}$ takes a wrong turn (i.e. there's an incorrect transition). Let e be the suffix of the counterexample γ from the point where the transition of $\hat{M}$ is inconsistent with the corresponding transition of the target M. Then e can be found by means of a search strategy such as binary search and membership queries.

As in the previous section, this section concludes with an analysis of the impact on the amount of membership queries needed and the space requirement.

Query complexity Rivest and Schapire's original improvement used a binary search strategy to find the single suffix e. As expected from a binary search, this results in $\mathcal{O}(\log(m))$ membership queries per counterexample of length m. The amount of equivalence queries is $\mathcal{O}(n)$, and thus the amount of membership queries due to counterexample analysis is $\mathcal{O}(n\log(m))$, where m is the size of the longest counterexample. The total amount of membership queries including hypothesis construction is thus $\mathcal{O}(kn^2 + n\log(m))$, an improvement over that of Kearns and Vazarani's (see section 2.1.1).

It is worthwhile to note that Isberner et al. analyzed and compared multiple search strategies by means of an abstract framework[26]. Their results show that the query complexities of the other search strategy remain the same as the original binary search strategy due to Rivest and Schapire, yet perform better when put into practice.

Space Complexity From Isberner's analysis[24], the space complexity due to the new counterexample analysis is better than that of L^*, however it is worse than that of Kearns and Vazirani's: $\mathcal{O}(kn^2 + nm)$.

In the next section we will show another version of the algorithm due to Isberner et al., which combines Kearns and Vazirani's discrimination tree, and Rivest and Schapire-style counterexample analysis.

actually write this section?

3 Variants

In the first few years following the introduction of the L* algorithm by D. Angluin [6], L* learning was only a theoretical exploration. The various improvements described in the previous chapter, such as *Classification Trees* section 2.1 and the *TTT* algorithm **??** made practical applications already more feasible.

In applying L* or improvements thereof to practical applications, there were still some practical limitations and restrictions that had to be addressed (Steffen 2011, Ch. 6 [38]). Some of these issues can be summarized as follows:

- Equivalence queries are generally undecidable for black boxes
- L* can only interact with regular languages, not with real systems
- The amount of required membership queries can grow very fast
- Membership queries might not be independent in practice, which imposes the requirement for a *reset* function
- Not all systems might support a *reset* function.

In order to address these limitations and restrictions, several variants of the L* algorithm have been proposed, each with the goal to solve such an issue. In this chapter we will further elaborate on some of these issues and the variants proposed to solve them.

3.1 Approximating *EQUIV (M)* queries with *MEMBER (w)* queries

The target machine on which the learning algorithm is applied is generally a black box, since the main motive for using a learning algorithm is to infer knowledge about some unknown system. Therefore equivalence queries can generally only be answered by exhaustively testing the inputs and outputs of a system. However, exhaustively testing a black box is undecidably hard.

In order to solve this issue model-based testing techniques have been used, such as *Chow's W-method* [11, 9] or the *WP-method*. These methods rely on approximating *EQUIV (M)* queries by using *MEMBER (w)* queries.

To further research in this area, the *ZULU* challenge [10] was introduced. The *ZULU* challenge asked participants to find a DFA corresponding to a certain system as accurately as possible, while imposing a restriction on the number of *MEMBER (w)* queries and disallowing *EQUIV (M)* queries completely.

add some more text and a fluid intro to Chow's method

3.1.1 Chow's W-method

According to [41], the maximum number of test sequences needed to converge to a correct approximation of the system under learning is bounded by $n^2 \times k^{m-n+1}$ when using Chow's W-method, and the maximum length of these sequences is $n^2 \times m \times k^{m-n+1}$, where n is the number of states in the provided automaton, m is the number of states in the correct automaton, and k is the size of the input alphabet.

By refining the test suites and the specification at the same time, and by using a divide and conquer approach, the size of the test is considerably reduced in comparison to defining the tests from the final specification[21]. This is especially beneficial because complex systems are usually created in multiple iterations. Unlike some other test selection methods the W-method does not require that the number of states is exactly the same between the implementation and protocol. The W-method generates tests that guarantee correctness of an implementation with a number of states below a certain upper bound.

The W-method generates input sequences that reach every state in the diagram, checks all the transitions in the diagram and identifies all the destination states and verifies them against their counterparts in the implementation[21]. To achieve this, the W-method constructs two sets of input sequences:

- A state cover $S \subseteq \Sigma^*$ that reaches every state in the final state machine, including the empty sequence to reach the initial state.

- A characterization set $W \subseteq \Sigma^*$ that has different outputs for at least one sequence in W for every pair of different states.

Given a specification P that can be modelled by an unknown finite state machine M, the only information the algorithm needs is an estimate of the maximum number of states of an unknown model for the specification. Suppose $d =$ estimated maximum number of states for P $-$ number of states in M. Naturally, it follows that $d \geq 0$. We also take $\Sigma[n] = \Sigma^n \cup \cdots \cup \{\epsilon\}$, making $\Sigma[n]$ a random word with a maximal length of n. For the W-method the testing suite $T = S \cdot \Sigma[d+1] \cdot W$. The idea behind the W-method is that $S \cdot \Sigma[1] = S \cup S \cdot \Sigma$, which is also called the transition cover of M, ensures that all the states and transition in P are also present in M, while $\Sigma[n] \cdot W$ ensures that M and P are in the same state after performing all the transitions.

In order to use the W-method there are four requirements. The machine must be minimal, completely specified, completely reachable, and have a fixed initial state. With some additional manipulation these four assumptions may be violated. The method consists of three steps:

1. estimation of the maximum number of states.
2. generation of test sequences.
3. verification of the responses.

By exploiting domain specific knowledge, the W-method can be adapted to reduce the number of test cases even further. When dealing with a networked application for example, there are usually no further possible transitions when a connection is closed by a remote machine. Exploiting such knowledge can greatly reduce the number of tests necessary[11].

3.2 Limiting the amount of membership queries

In a theoretical framework the learning algorithm doesn't have to account for execution times of individual *MEMBER (w)* queries. In practice, such queries might take time or be expensive to execute. Therefore algorithms benefit from reducing the amount of queries needed. Some generic improvements in this regard have already been covered in section 2.

However, when using for example *Chow's method* for *EQUIV (M)* queries, the amount of membership queries needed grows exponentially in the number of states section 3.1.1.

Cite ZULU participant here, since the challenge also imposed a maximum on *MEMBER (w)* queries

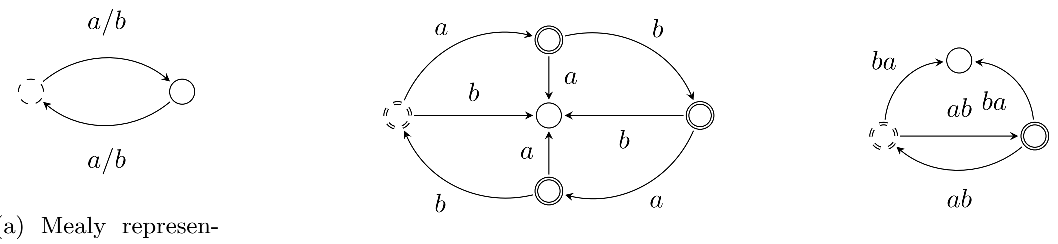
3.3 Learning Mealy machines

The L^* algorithm of Angluin is able to learn DFA's, but it can not directly learn mealy machines. A mealy machine, as opposed to a DFA, has no accepting states. Instead, mealy machines additionally define an output symbol on each state transition. Reactive systems, or I/O programs, are easily modeled by a mealy machine, but not as easily by a DFA. Therefore, some adaptations are required. There are two main ways to do this. Either the mealy machine can be transformed into a DFA, or an adaptation of L^* called L^*_M can be used to learn the mealy machine directly.

3.3.1 Transforming a mealy machines into a DFA

Say we have a mealy automaton M with input alphabet Σ_M and output alphabet O that we want to transform into a DFA D with input alphabet Σ_D. There are two ways to do this.

The first first way is to define $\Sigma_D = \Sigma_M \cup O$ [20]. This transformation defines that each state in M is also a state in D. For each transition in M from state q_a to q_b constrained by i and output symbol o, a new intermediate state q_t is introduced in D. Two transitions are added: one from q_a to q_t constrained by i, and one from q_t to q_b constrained by o. Furthermore, a single error state k is added to D, which is the only non-accepting state. For each combination of a node q and input



(a) Mealy representation

(b) Hungar's transformation

(c) Mäkinen's transformation

Figure 2: Mealy to DFA transformation

symbol $\sigma \in \Sigma_D$, if q does not yet contain a transition constrained by σ, a transition constrained by σ is added from q to k. An example of this transformation can be seen in figure 2b.

The second way to transform M is to define $\Sigma_D = \Sigma_M \times O$ [30]. In other words, Σ_D contains all combinations of input and output symbols from M. This removes the need for intermediate states, but can have a significant impact on the size of the input alphabet. Similarly to the previous transformation, one error state is defined and all other states are accepting states. If from a certain state an input/output combination does not occur, then a transition to the error state is added with that combination. An example of this transformation can be seen in figure 2c.

Either of these transformations can be used to learn a mealy machine with the standard L^* algorithm.

for both of these, explain how membership queries are defined

To learn a mealy machine M, simply define Σ as $\Sigma_M \cup O$ or $\Sigma_M \times O$. By doing this, a model D is learned which is the transformed version of M.

3.3.2 Adapting L^* into L^*_M

Adapting Angluin's L^* algorithm has been adjusted for use with mealy machines. The new procedure was first informally described in [32], and later more rigorously in [36]. In L^* the membership queries return a boolean value; either it's a member, or it's not. For L^*_M, the membership query is replaced by a output query, which returns an output string. The observation table is modified to store these output strings instead of boolean values. As in L^*, output queries are used in order to fill in the observation table. Suppose the algorithm performs an output query for $s \cdot e$, where $s \in S \cup (S \cdot \Sigma)$ and $e \in E$. The query will produce a string r with $|r| = |s| + |e|$, where $|x|$ denotes the length of the string x. Instead of storing r directly in the observation table, only the last $|e|$ symbols of r are stored (since the observation is prefix closed, the output corresponding to s is already stored in the table). The intuitive meaning of this value is that it corresponds to the sequence of output symbols that is produced when the when input e is applied from the state corresponding to s.

With the new meaning of the values in the table, a value of ϵ in E becomes nonsensical. This is because ϵ would denote no change in state, so there would be no output symbol. Therefore, at the start of the algorithm, E is not intialized to $\{\epsilon\}$ but to Σ.

The concepts of closedness and consistency remain identical. The procedure to build the hypothesis remains largely the same, but now the generated hypothesis no longer defines accepting states, but instead defines output strings on the transitions.

note that mealy dfa

3.4 Systems without a *reset* function

In the L* learning algorithm, every *MEMBER (w)* query is implicitly independent of other queries. However, when learning a system in practice, it might happen that *MEMBER (w)* queries are not independent at all. Suppose for example that the system under learning requires authentication while also imposing a restriction on the maximum amount of failed requests. In such a system queries are no longer independent of each other, since after a certain amount of authentication requests the systems behaviour suddenly changes for the same membership queries. In order to solve this issue, the L* algorithm implicitly requires a means of *reset*ting the system under learning (Rivest & Schapire, pg. 301)[35].

However, not all systems support a *reset* function. As we'll see in section 6, the system under learning could also be a system that the learning algorithm has totally no control over.

Explain homing sequences here

4 From theory to practice

As the previous chapters have shown, active state machine learning is a well studied theoretical learning technology. Slowly but surely activate state machine learning is moving from a theoretical novelty to a real world applicable technology, due to hardware advancements as well as the optimizations discussed in chapter section 2.

The learning algorithms discussed in the previous chapters all required an (abstract, formal) input alphabet, an (abstract, formal) output alphabet, membership queries and equivalence queries. In practice, it is in many cases quite difficult to realize these requirements for a specific system under learning[39]. This section discusses these challenges faced when applying active state machine learning to real world scenarios.

Interaction with real world applications Before active state machine learning can be used in practice, a mapping has to be made from an abstract, formal input alphabet to a concrete real world "language" understood by reactive systems (it is evident that reactive systems are a requirement for practical active state machine learning)

begrippenlijst aanmaken met definities?

. This mapping needs to result in a deterministic language for it to work[39]. In the same way, the output of the system under learning needs to be mapped back to an abstract language understood by the tool (see section section 5 below).

Whilst such a mapping has different peculiarities per application, there is an overlapping theme that applies to every mapping: it needs to be abstract enough in terms of communication (leading to a useful model structure, see below) while allowing for an automatic back and forth translation between the abstract and the concrete languages[39]. An interesting fact is that it turns out that the active state machine learning algorithm can be enhanced per application using application-specific optimizations[20]. For more information on recent work focusing on these abstractions, see [4, 18, 27].

Besides mapping input- and output alphabets, the gap between the abstract learned model and the concrete application also has to be bridged: when an abstract learned model is presented to the user, it has to be "translated" to a representation of the system under learning. This is rather intuitive for those applications that are designed explicitly for connectivity (such as web services or communication protocols), because these are made to be invoked from the outside. For other types of applications, this can be arbitrarily difficult.

Another obstacle to overcome is that of parameters used in real world applications. Think, for example, about increasing sequence numbers in communication protocols. This is still a huge challenge to resolve[39], and, for now, besides the creation of prototypical solutions[4, 37, 17], application-specific solutions have to be applied.[39].

The final hurdle is that of a "reset". Membership queries have to be independent (see chapter section 3.4). In practice this often means that applications have to be reset in between successive membership queries. This can be achieved using homing sequences[35] (discussed in chapter section 3 or by simply restarting the application for each membership query.

Membership queries Membership queries are the most straightforward to translate to real world applications: they can be realized via testing. An important thing to note, however, is the amount of membership queries required. Learning a real world application can easily require several thousand membership queries

Add citation

. This means that the time required to learn a model can be greatly reduced either by speeding up membership queries (executing them in parallel), or simply by reducing the number of membership queries required. Chapter section 2 has discussed several improvements to achieve exactly that. Additionally, application-specific optimizations can be used to further reduce the number of membership queries required.

Equivalence queries In theoretical simulations, equivalence testing is often easy because often the target system (in some cases even the model!) is known. In practice, however, the system under test is usually a black box system. This means that equivalence queries will have to be approximated, typically using membership queries. These approximated membership queries are in general not decidable without assuming any extra knowledge[39], such as the number of states of the system under learning: it is impossible to be certain that the system has been tested extensively enough.

An alternative to using membership queries to simulate equivalence queries, is to use model-based testing methods[8, 40]. An example of model-based testing is Chow's W-method[9] (discussed in chapter section 2), which can be used if an upper bound on the number of states in the system is known.

Mention Wp-method[12] and more examples? Or refer to chapter section 3

Discuss first paragraph of page 33 in [39]?

All the above problems might give the impression that active state machine learning is not yet applicable to real world applications. While it is true that practical application is not yet complete, great progress has been made and great things have already been achieved. The next few highlights of real world applications serve to illustrate these facts.

5 Tools

In order to facilitate and promote active automata learning in a practical setting, tools have been created. This section will give a brief overview of two tools. First, the tool Learnlib (5.1) will be discussed. This tool provide various active automata learning algorithms and optimizations. The second tool is Tomte(5.2), which automatically generates abstractions for automatons in order to apply active automata learning on them. This selection is based on the importance of the tool, availability and the amount of researches that have used those tools.

5.1 Learnlib

Learnlib [2] is a library that implements various active learning algorithms as well as different configurations for learning automata. It has been in development since 2009 [34] and as of 2015, there has been a total overhaul of the tool [25]. To avoid confusion, the old version was renamed to JLearn.

The current version of this tool consists out of two parts: Automatalib and Learnlib.

Automatalib An independent library that contains abstract automata representations, automata data structures and algorithms. The abstract automata representations make the library flexible because all data structures and algorithms depend on those representations. This makes it easy to add third party implementations of automata like the BRICS library [5]. Automatalib

[2]Supported through bug fixes and available from `https://github.com/LearnLib/learnlib`

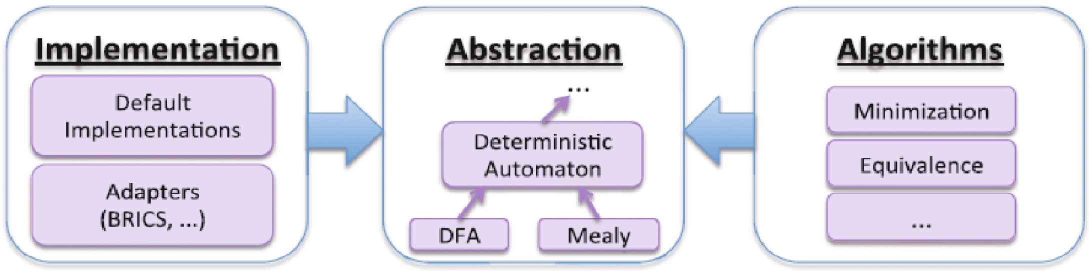


Figure 3: Architecture of Automatalib, source is from [25]

includes minimalization algorithms and equivalence testing algorithms based on Hopcroft and Karp's near-linear equivalence algorithm [14] or the W-Method (X)

Need a reference to the W-Method

for black-box testing.

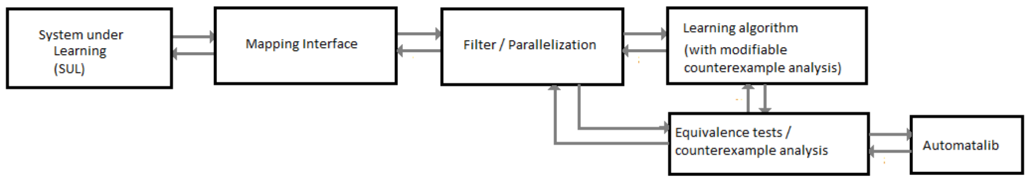

System under Learning (SUL)
Mapping Interface
Filter / Parallelization
Learning algorithm (with modifiable counterexample analysis)
Equivalence tests / counterexample analysis
Automatalib

Figure 4: Architecture of Learnlib

Learnlib A library that provides learning algorithms and infrastructure for automata learning. The learning algorithms consist of a "base algorithm" whereby the counterexample analysis can be exchanged with other methods. All the different base algorithms with the examples of variants that are officially supported are listed below:

- L* (base) (Explained in section 1)
  - Maler & Pnueli's [31]
  - Rivest & Schapnire's (Summarized in
  - Shahbaz's [36]
  - Suffix1by1 [22]
- Obversation Pack (base) [19]
- Kearns & Vazirani's (base) (Summarized in
- DHC (base) [33]
- TTT (base)
- NL* (base) [7]

All the algorithms come with both DFA and Mealy versions, expect for DHC and NL*.
For finding the counterexamples, Learnlib uses Automatalib as well as other methods like randomized tests. More methods can be found in [25].

Learnlib also offers filters for reducing the amount of queries such as elimination of duplicate queries. contains a parallellization component that can speed up the process by using multiple teachers and parallel execution of membership queries[13][15].

5.2 Tomte

Tomte [3] is a tool that automatically makes abstractions for automata learning. Essentially, it a connector between the system under learning(SUL) and the learner. This makes using Learnlib and Libalf easier, since the user doesn't have to make the mapping.

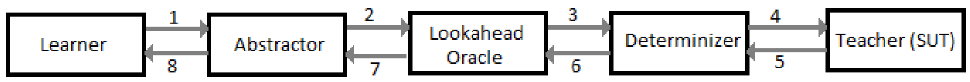


Figure 5: Architecture of Tomte

The Abstractor, Lookahead Oracle and the Determinizer together form Tomte. The other two parts are not part of Tomte but it comes with a supplied library (Learnlib) for making the learner. The makers also have a tool [4] available for creating the SUL since they must be modeled after a register automata.

Determinizer The Determinizer elimates the nondeterministic behaviour caused by the SUL. Since tools like Learnlib can only analyse deterministic behaviour, it needs to converted. The theory behind it, is explained in [2].

Lookahead Oracle This oracle is used to annotate events of the SUL with information about the impact on the future behaviour of the SUL. This way, the Looahead Oracle act as a cache for the Abstractor. The theory and implementation of this oracle is found in [3] and [1].

Abstractor The Abstractor is the component that creates the mapping between the SUL and the learner. The idea behind the Abstractor is to make an abstraction of the parameter values of the SUL but leaving the input/output symbols unchanged. It uses counterexample-guided abstraction refinement[1] for extension of the mapping. In order to make it scalable, this component also tries to reduce the length of the counterexample by removing loops and single transitions [29]. The complete theory is found in [1].

Interaction between the modules The exchange of messages between all the modules is as follows(see numbering in figure 5.

1. Learner sends an abstract output query to the Abstractor.
2. Abstracter receives an abstract query, selects a concrete input symbol i and send it as an output query to the Lookahead Oracle. If no input symbol i exists, return $\perp$ to the Learner.
3. Lookahead Oracle checks if i is cached. If not, then s is send to the Determinizer. Otherwise, go to step 7.
4. Determinizer transforms the input back to the original behavior of the Teacher
5. Determinizer receives input from the Teacher and transforms this nondeterministic behavior.
6. Determinizer sends concrete symbol o to the Lookahead Oracle which caches input output pair $\{i, o\}$

[3]In active development and available from `http://tomte.cs.ru.nl/Tomte-0-4`
[4]SUL Tool available from http://tomte.cs.ru.nl/Sut-0-4/Description

7. Lookahead oracle sends o together with the annotated information that came after the sequence of queries between the last reset query and i.

8. Abstractor receives a concrete answer o and the annotated information. It uses the annotated information to determine updates to the state variables of the Learner. It sends those updates and other information to the Learner as an abstract answer.

6 Applications

This chapter serves to highlight some interesting and important practical applications of active state machine learning. The goal is not to discuss individual applications, rather several "domains" of application have been identified. Before doing so, however, the bridge between theory and application must be crossed.

Conclusion

References

[1] Aarts. *Tomte: Bridging the Gap Between Active Learning and Real-world Systems.* PhD thesis, 2014.

[2] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. *Theoretical Aspects of Computing - ICTAC 2015: 12th International Colloquium, Cali, Colombia, October 29-31, 2015, Proceedings*, chapter Learning Register Automata with Fresh Value Generation, pages 165–183. Springer International Publishing, Cham, 2015.

[3] Fides Aarts, Falk Howar, Harco Kuppens, and Frits Vaandrager. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, chapter Algorithms for Inferring Register Automata, pages 202–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[4] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Testing Software and Systems*, pages 188–204. Springer, 2010.

[5] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, January 2005.

[6] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[7] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 1004–1009, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[8] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*, volume 3472. Springer, 2005.

[9] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 05 1978. Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) May, 1978; Last updated - 2011-07-20; CODEN - IESEDJ.

[10] David Combe, Colin Higuera, and Jean-Christophe Janodet. *Finite-State Methods and Natural Language Processing: 8th International Workshop, FSMNLP 2009, Pretoria, South Africa, July 21-24, 2009, Revised Selected Papers*, chapter Zulu: An Interactive Learning Competition, pages 139–146. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[11] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.

[12] Susumu Fujiwara, Gregor V Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*, 17(6):591–603, 1991.

[13] Marco Henrix. Performance improvement in automata learning. Master's thesis, 2015.

[14] J.E. Hopcroft and R.M. Karp. *A linear algorithm for testing equivalence of finite automata*. Technical report (Cornell University. Dept. of Computer Science). Defense Technical Information Center, 1971.

[15] Falk Howar, Oliver Bauer, Maik Merten, Bernhard Steffen, and Tiziana Margaria. *Leveraging Applications of Formal Methods, Verification, and Validation: International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, chapter The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning, pages 232–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[16] Falk Howar, Malte Isberner, and Bernhard Steffen. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, chapter Tutorial: Automata Learning in Practice, pages 499–513. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[17] Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. On handling data in automata learning. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 221–235. Springer, 2010.

[18] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In *Verification, Model Checking, and Abstract Interpretation*, pages 263–277. Springer, 2011.

[19] Falk M Howar. *Active learning of interface programs*. PhD thesis, 2012.

[20] Hardi Hungar, Oliver Niese, and Bernhard Steffen. *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings*, chapter Domain-Specific Optimization in Automata Learning, pages 315–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[21] F. Ipate and L. Banica. W-method for hierarchical and communicating finite state machines. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 2, pages 891–896, June 2007.

[22] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. Angluin Style Finite State Machine Inference with Non-optimal Counterexamples. In *1st International Workshop on Model Inference In Testing, MIIT 2010*, pages 11–19, Trento, 2010.

[23] Malte Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, 2015.

[24] Malte Isberner, Falk Howar, and Bernhard Steffen. *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, chapter The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning, pages 307–322. Springer International Publishing, Cham, 2014.

[25] Malte Isberner, Falk Howar, and Bernhard Steffen. *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, chapter The Open-Source LearnLib, pages 487–495. Springer International Publishing, Cham, 2015.

[26] Malte Isberner and Bernhard Steffen. An abstract framework for counterexample analysis in active automata learning. In *ICGI*, pages 79–93, 2014.

[27] Bengt Jonsson. Learning of automata models extended with data. In *Formal Methods for Eternal Networked Software Systems*, pages 327–349. Springer, 2011.

[28] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[29] Pieter Koopman, Peter Achten, and Rinus Plasmeijer. *Trends in Functional Programming: 14th International Symposium, TFP 2013, Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, chapter Model-Based Shrinking for State-Based Testing, pages 107–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[30] Erkki Mäkinen and Tarja Systä. Mas &mdash; an interactive synthesizer to support behavioral modelling in uml. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[31] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316 – 326, 1995.

[32] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT*, pages 95–100. IEEE Computer Society, 2004.

[33] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. *Leveraging Applications of Formal Methods, Verification, and Validation: International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, chapter Automata Learning with On-the-Fly Direct Hypothesis Construction, pages 248–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[34] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer*, 11(5):393–407, 2009.

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

[36] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 207–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In *Testing of Software and Communicating Systems*, pages 319–334. Springer, 2007.

[38] Bernhard Steffen, Falk Howar, and Maik Merten. *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, chapter Introduction to Active Automata Learning from a Practical Perspective, pages 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[39] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.

[40] Jan Tretmans. Model-based testing and some steps towards test-based modelling. In *Formal Methods for Eternal Networked Software Systems*, pages 297–326. Springer, 2011.

[41] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9:653–665, 07 1973.