Karlsruhe Institute of Technology

Learning-based software testing: Evaluation of Angluin's L* algorithm and adaptations in practice

Bachelor Thesis of

Maximilian X. Czerny

At the Department of Informatics
Institute for Theoretical Computer Science

Reviewer:	Junior Prof. Dr. Mana Taghdiri
Second reviewer:	Prof. Dr. Karl Meinke
Advisor:	Dr. Alexander Kozlov

Duration:: 22. January 2014 – 21. May 2014

Abstract

Learning-based testing (LBT) can ensure software quality without a formal documentation or maintained specification of the system under test. For this purpose, an automaton learning algorithm is the key component to automatically generate efficient test cases for black-box systems. In the present report, Angluin's automaton learning algorithm L* and the extension called L* Mealy are examined and evaluated in the application area of learning-based software testing. The purpose of this work is to evaluate the applicability of the L* algorithm for learning real-life software and to describe constraints of this approach. To achieve this, a framework to test the L* implementation on various deterministic finite automata (DFAs) was written and an adaptation called L* Mealy was integrated into the learning-based testing platform LBTest. To follow the learning process, the queries that the learner needs to perform on the system to learn are tracked and measured. The main results of this thesis are that (1.) L* shows a near-linear learning trend in the state space size of the DFAs for easy-to-learn automata, (2.) even for hard-to-learn DFAs the algorithm performs better than the theoretical predictions imply, (3.) L* Mealy shows a polynomial growth on the membership queries during the learning process and (4.) during the learning process L* and L* Mealy rarely built a hypothesis which makes L* Mealy inefficient for LBT.

Acknowledgements

First and foremost, I am obliged to thank my research supervisors Prof. Karl Meinke and Dr. Alexander Kozlov. Their dedicated support helped me to accomplish this thesis. In August 2013, I went to the Royal Institute of Technology (KTH) for one year. The presented research is owed to my time there at the School for Computer Science and Communication (CSC). I thankfully acknowledge the role of my reviewer Junior Prof. Mana Taghdiri at the Karlsruhe Institute of Technology (KIT) who enabled me to write this report abroad.

I thank the College for Gifted Students of the Department of Informatics at KIT for giving me a scholarship for this thesis. I gratefully acknowledge the support of the ARTEMIS project 269335 MBAT which enabled me to evaluate practical examples in Section 4.2.

Contents

Abbreviations

DFA	deterministic finite automaton
SUT	system under test
LBT	learning-based testing
CC	Cruise Controller
BBW	Brake-By-Wire

1. Introduction

As automated systems grow in size (see for example [DR08]) and penetrate our daily life starting with simple gadgets and growing into highly safety critical applications, such as car controllers, both testing time and efforts grow rapidly. Although these systems are capable of facilitating our life, without profound confidence in their reliability the actual execution might range from unsatisfying to dangerous — e.g. in an automotive ABS system. Additionally, the occasional lack of a formal and maintained specification for existing software — not to speak of opaque hardware components — challenges machine and exhaustive human-based testing. Therefore, there is a need for intelligent and especially automated testing that copes with a variety of different systems thus, avoiding error-prone manual evaluation.

The idea of automated learning-based black-box testing, as described by Meinke et al. [MNS12], is impressive because of the minimal logical efforts it takes developers to verify the system under test (SUT) even without a complete specification [NP10]. In order to automatically generate test cases, a learning algorithm can be applied to a reactive system. Reactive systems are especially suited for an automated testing approach and have been the subject of research for decades. Among these systems are for instance sensor-reactor machines, control units, web services and various communication protocols. The most distinguishing property is that these programs continuously execute on a set of input values and publish their results using outputs. To be applicable for automated learning, the SUT must be able to read sequences of input values and to eventually signal the outcome of execution using a sequence of output values.

There are various ways of modelling reactive systems such as deterministic finite automata (DFA) [Sip06], Mealy automata [Mea55] or Kripke structures as explained in [SB04]. Chapter 2 will cover the preliminaries needed for this report and describes the automata that were used. As first shown by Dana Angluin in [Ang87] using machine-learning techniques, it is possible to fully learn a DFA in polynomial time without knowing the actual representation by actively asking questions to an oracle. This approach is called the L* algorithm and will be described in Section 2.1. As many reactive systems go beyond binary in- and output data a DFA presentation thereof might not be adequate. To solve this problem a Mealy automaton, as defined in [Mea55], can be introduced. Both vectors of input values and output values can be handled with only minor adaptations to the L* algorithm — then called the L* Mealy algorithm which will be described in Section 2.2. Finally, a practically useful approach to automated learning-based testing called LBTest, which will be described in Section 2.3 and was discussed by Meinke and Sindhu in [MS13], made use of the automata-learning algorithm L* Mealy to derive efficient test cases for the SUT. Thus, it will be demonstrated that LBTest is able to learn any executable reactive system — regarded as a black box — and automatically test it without the need for formal documentation.

The purpose of this thesis is to collect and analyse empirical data on the performance of the L* learning algorithm and to discuss the observations with a specific focus on the theoretically deduced performance predictions made by Angluin [Ang87]. Furthermore, the applicability of the L* Mealy algorithm in learning-based testing ought to be assessed using case studies.

The structure of this report is as follows. In Chapter 2 the learning algorithms L* and L* Mealy will be introduced and the approach of LBTest will be explained. Chapter 3 will investigate the implemented testing framework for L* and the integration of L* Mealy into LBTest. In Chapter 4 the evaluation will be conducted and the gathered data will be presented. Chapter 5 will discuss the observational results and possible optimizations. Finally, Chapter 6 will give conclusions and possible directions for further research.

2. Foundations and related work

To understand this work, it is essential to be familiar with formal languages, prefixes and suffixes as covered by Rozenberg and Salomaa [RS09]. Further elementary knowledge, such as Nerode's right congruence and equivalence classes, specific to the area of automata theory is presented by Berg et al. [Ber+03].

Narendra and Thathachar [NT74] define a learning system by the ability to improve the behaviour with time towards an eventual purpose. In the case of L* and L* Mealy, the goal is behavioural equivalence between the learned automaton and the initially unknown automaton. Therefore, the learner actively experiments with the automaton output to detect equivalence classes in the input sequences. The learning algorithms can generate a hypothetical automaton from these equivalence classes as shown in [Ber+03].

To apprehend the learning process it is necessary to know how a system can be represented as an automaton. There are various possibilities of which a deterministic finite automaton (DFA) is the most basic. Sipser [Sip06] defines a DFA thus:

$$A_{\text{DFA}} = (Q, \delta, q_0, F). \tag{2.1}$$

Q is a set of states, $\delta : Q \times \Sigma \to Q$ is a transition function that maps a state $q_{current} \in Q$ and a symbol $a \in \Sigma$ from an alphabet to the next state $q_{next} \in Q$, $q_0 \in Q$ is the start state and $F \subseteq Q$ the set of all accepting states.

Since DFAs are very simplistic models of computation, Mealy [Mea55] defined a more general automaton model thus:

$$A_{\text{Mealy}} = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F). \tag{2.2}$$

The main extensions in this model are the set of output symbols Ω and the output function $\lambda : Q \times \Sigma \to \Omega$ which maps each state $q \in Q$ and each symbol $a_{in} \in \Sigma$ to an output symbol $a_{out} \in \Omega$.

2.1. The L* algorithm

In machine learning theory one differentiates between *active* and *passive* learners. While active learning involves using queries to retrieve observations, passive learning is applied to an existing set of labelled data. It was proven by Gold [Gol78] in 1978 that the latter approach is NP-hard. Therefore, it was a huge breakthrough when Dana Angluin introduced the active L* learner [Ang87] and formally deduced its polynomial runtime in 1987. Ever since, her algorithm has been used as a paramount example in teaching machine learning.

The purpose of Angluin's L* algorithm is to gather enough knowledge about an unknown DFA to formulate a hypothetical representation of it from the observed behaviour. It is capable of deciding itself what additional data it requires to fully learn another system and to fetch the information through a clearly defined interface from an external representation. The L* algorithm as described by Angluin [Ang87], Chen [Che13] and Berg et al. [Ber+03] consists of several components depicted in Figure 2.1 which interact in order to resolve the structure of an initially unknown deterministic finite automaton.

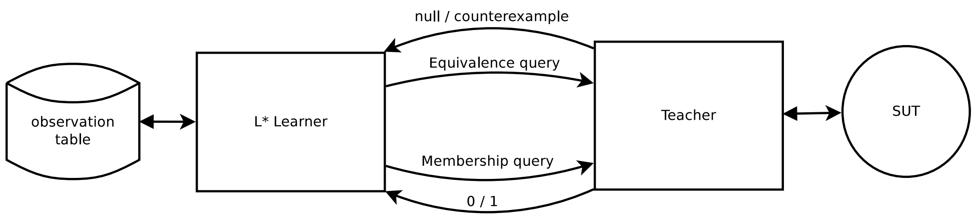


Figure 2.1.: Components of the L* Algorithm

Components

(1) A *learner* that controls the learning progress by deciding which questions to ask the teacher (2).

(2) A *teacher* that responds to the informational needs of the learner with data from the SUT (3).

(3) An *SUT* that can be represented by an automaton.

(4) An *observation table* that holds all the information that the learner (1) has about the SUT so far. If the learner finishes, it must be possible to create a behaviourally equivalent system to the SUT from the data.

The learner is the core component that controls the learning progress and decides what to investigate. Assuming that an unknown system can be modelled as a DFA $A = (Q, \delta, q_0, F)$, the learner L^* identifies the accepted language $L(A)$. The actual learning is a simple mapping of strings from the input alphabet Σ to $\{0, 1\}$ depending on whether the automaton A accepts (1) or rejects (0) the input strings. The automaton accepts an input sequence of symbols if the resulting state q_{res} according to the transition function δ is an accepting state $q_{res} \in F$. In theoretical terms, the learner's task is to identify all equivalence classes of input strings Σ^*. This can be achieved using properties of the Nerode right congruence and is explained in depth by Berg et al. [Ber+03].

As the learner itself does not possess any knowledge about the structure of the unknown system, it has to retrieve information by creating queries for missing data and sending them to a teacher T. After all, the teacher component knows the internal structure of the system that should be learned — i.e. it knows A. Thus, the learner is capable of asking questions to the teacher in order to gather data on the unknown system without revealing the structure at any point:

Queries

$Aut(\Sigma)$ is defined as the set of all DFAs (see Definition 2.1) with input alphabet Σ.

(i) *Membership queries* $Q_M : \Sigma^* \rightarrow \{0, 1\}$:
The learner L^* may ask the teacher T to perform a test run on A to check for a given string x if $x \in L(A)$ then T replies with 1 or if $x \notin L(A)$ then T replies with 0. Using Q_M, the learner can build a data structure from which it can derive A.

(ii) *Equivalence queries* $Q_E : Aut(\Sigma) \rightarrow \Sigma^* \cup \{true\}$:
The learner L^* may ask the teacher T to check whether the current learning outcome $A' \in Aut(\Sigma)$ from the set of all possible outcomes $Aut(\Sigma)$ is equivalent to the hidden automaton A. As the learner performs a series of membership queries it fills in missing knowledge and becomes confident about the correctness of the learned data. Thus, it assembles a DFA A' from previously gathered data and requests an equivalence check. The learner performs the equivalence check as described by Norton [Nor09] on A' and A. If it yields a counterexample $Q_E(A') = c \in \Sigma^*$ to the equivalence of A' such that $c \in L(A) \wedge c \notin L(A')$ or $c \notin L(A) \wedge c \in L(A')$, the teacher returns c and the learner incorporates it which can then trigger further membership queries. Should no such c exist ($Q_E(A') = true$), the L^* algorithm has successfully learned A.
In this role, the teacher acts as an oracle because it has to know the representation of A to perform the equivalence check. Consequently, this approach does not work with black-box testing.

To direct the learner's progress Angluin defines two properties in [Ang87] that L* should validate the gathered data against. If one of the properties does not hold against the observations so far, further membership queries are needed to inspect a certain area of the hidden automaton.

Observation table properties

(a) An observation table is defined as *complete* if for each state in the hypothesis and each symbol from the alphabet Σ there is a clearly determined next state in the transition function δ. A complete table ensures that the learner can construct a DFA with a well-defined transition function δ — i.e. there is a transition from each state for each symbol.

(b) An observation table is defined as *consistent* if all strings in the same equivalence class that start in one state result in the same target state for transitions using any symbol $s \in \Sigma$. The consistency property ensures that the learner differentiates between the learned equivalence classes and splits them if any discrepancy is detected in the hypothesised right congruence of input strings from Σ^*.

In conclusion, the table properties determine which membership queries the learner is supposed to perform. Once the observations are complete and consistent the learner can create a hypothesis about the DFA which can be assured to be a connected graph with all equivalence classes that L* could derive. This automaton is then handed to the learner for an equivalence check with the correct DFA using an equivalence query. For a formal definition of consistency and completeness see Section 3.1.3.

The result of repeating this procedure until an equivalence query does not return a counterexample, is an algorithm that learns arbitrary DFAs in polynomial time [Ang87]:

$$O(m|\Sigma||Q|^2), \tag{2.3}$$

where m is the length of the longest counterexample returned by the teacher, Σ is the input alphabet and Q is the set of states of the DFA to be learned. The L* algorithm is an example of a *complete* learner which means that the emphasis is on terminating with the final result rather than producing many intermediate incremental results. Section 4.1.1 will show how the algorithm successfully learns an exemplary initially unknown automaton.

2.2. The L* Mealy algorithm

In practice software output goes beyond the binary values of 1 ($true$) and 0 ($false$) that the deterministic finite automata can cope with. Thus, complex systems require another more practical representation for modelling such as the previously introduced Mealy automaton. Mealy automata are able to process arbitrary and discretionary input vectors, processes the values and return arbitrary and discretionary output vectors.

The L* Mealy algorithm, as described by Niese [Nie03], is an adaptation of Angluin's L* algorithm as previously introduced in Section 2.1. The main difference is that the system is represented as a more general Mealy automaton. The algorithmic concepts remain whereas the data structures change. The observations in L* Mealy are vectors but are treated similar to the binary information in L*. The components exchange the data in a slightly different way which is based on the Definition 2.2 of the Mealy automaton $A_{\text{Mealy}} = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$.

Queries

$Mealy(\Sigma, \Omega)$ is defined as the set of all Mealy automata with input alphabet Σ and output alphabet Ω.

(i) *Membership queries* $Q_M : \Sigma^* \to \Omega^*$:
As opposed to membership queries in L* the learner queries the informational needs to a teacher that uses a Mealy automaton as representation of the hidden system. Thus, each response is a vector $o \in \Omega^*$.

(ii) *Equivalence queries* $Q_E : Mealy(\Sigma, \Omega) \to \Sigma^* \cup \{true\}$:
The only difference to the equivalence queries in L* is the format of the learner's hypothesis $A'_{\text{Mealy}} \in Mealy(\Sigma, \Omega)$ which is a Mealy automaton. The counterexample $Q_E(A'_{\text{Mealy}}) = c \in \Sigma^* \cup \{true\}$ is processed in the same way as L* would do it.

In contrast to the queries, the completeness 2.1.(a) and the consistency property 2.1.(b) are specified in the same way as for L*. Using this advanced approach, it is possible to learn automata of real-life complexity as the next section shows.

2.3. LBTest

There are various algorithms and tools that can be applied to learning-based testing as discussed by Sindhu [Sin13]. This report focusses on the LBTest tool [MS13] which uses black-box learning in order to automatically test software components. It builds upon the early work on automata learning, model checking and testing with a particular focus on L* by Groce et al. [GPY06]. As opposed to the previous learning algorithms in Section 2.2 and Section 2.1 that had a representation of the actual system at hand, LBTest learns the software without knowledge of its structure. The learning-based testing (LBT) approach, as described by Meinke et al. [MNS12], uses a set of requirements to validate the given software against, a specification of discrete input and output variables and their types. Thus, LBT can be easily integrated into the software development cycle or can be applied to a finished system without any knowledge of the internal structure apart from a tiny input and output interface. Conceivably, the software specification could be automatically generated during the development stage and the software user requirements could be mechanically translated from a textual project description. This makes LBT a smooth approach to ensure software quality.

In Figure 2.2 the architecture of LBTest with adaptations from [MS13] is depicted.

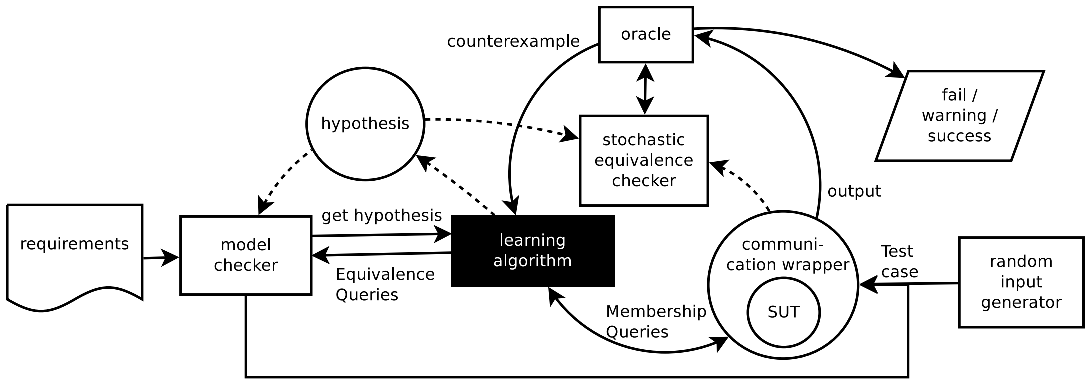


Figure 2.2.: The architecture of the LBTest software in which the learning algorithm is embedded

Architecture

The core components of the LBTest architecture are:

(1) A *model checker* as described in [GA07] which can formally validate a given model against a specification (which in the case of LBTest is a temporal logic). Using a hypothesis about the learner and a set of rules for validation, the model checker can generate efficient test cases for the SUT as counterexamples to requirements.

(2) The *user requirements* are used by the model checker to validate a given system. These logical rules are highly expressive and can state behavioural requirements of a system.

(3) The *learning algorithm* is able to discover the internal structure of the SUT (4). For this purpose, the L* Mealy algorithm, as introduced in Section 2.2, will be used in this work.

(4) The *system under test* is a black-box software or hardware component that should be tested against some requirements (see (2)). The learning algorithm e.g. L* Mealy might need to discretise the input and output vectors of the SUT. For this purpose a communication wrapper is needed around the SUT.

(5) A *random input generator* can generate random test cases for the SUT for stochastic equivalence checking.

(6) An *oracle* compares the results of the SUT to the expected values by the learner's hypothesis. Should an inconsistency arise, the learner is given a counterexample to the hypothesis and has to incorporate it. If the SUT does not comply with the intended behaviour, the oracle can return warnings or errors that indicate a problem in the SUT.

(7) A *stochastic equivalence checker* is used to compare a learned system to the actual system (4). It is able to guarantee the equivalence of the two automata with some probability.

The definition of the queries is consistent with the definition of the queries used by L* Mealy in Section 2.2. However, they trigger different actions in the LBTest tool. Compared to the previous setup, in which a teacher could simply look into the hidden system (for further reference this will be called white-box learning), the SUT is a black-box.

Queries

(i) *Membership queries* $Q_M : \Sigma^* \rightarrow \Omega^*$:
These are performed by invoking the SUT with the current vector of input values $i \in \Sigma^*$ and retrieving the output values $o \in \Omega^*$ as a result. Since the input is handled externally by the SUT (4), this process can be considerably time-consuming depending on the complexity of the SUT.

(ii) *Equivalence queries* $Q_E : Mealy(\Sigma, \Omega) \rightarrow \Sigma^* \cup \{true\}$:
Equivalence queries cannot be answered by a teacher as in 2.1.(ii) because the internal structure of the SUT is hidden. Instead stochastic equivalence checking is used and the counterexamples come from the random input generator in the LBTest environment. Once the learner has created a complete and consistent set of observations using membership queries, a hypothesis $A'_{\text{Mealy}} \in Mealy(\Sigma, \Omega)$ that is represented as a Mealy automaton is generated from the data. The model checker (1) then validates the automaton against the specified requirements (2). If any flaw is detected, the model checker generates a test case $Q_E(A'_{\text{Mealy}}) = c \in \Sigma^*$ for the SUT (4). Thereafter, the oracle (6) is able to compare the assumed results $c_{ass} \in \Omega^*$ from the hypothesis with the actual values $c_{corr} \in \Omega^*$ from the test run. Should a discrepancy $c_{acc} \neq c_{corr}$ arise, the hypothesis was wrong and then the test case c is used as a counterexample for the learner. If $c_{acc} = c_{corr}$, LBTest found an inconsistency between the requirements and the SUT which returns a failure and aborts the execution. On the other hand, if the model checker cannot find any invalid sequence c to the hypothesis A'_{Mealy}, the random input generator (5) is consulted. It will iteratively generate a random test sequence which will then be used to stochastically check the equivalence between the hypothesis automaton and the SUT. If the stochastic equivalence checker (7) considers the hypothesis to be approximately probably correct $Q_E(A'_{\text{Mealy}}) = true$, no additional counterexamples are needed — i.e. the learning is finished without contradicting the correctness of the given system so it is likely to coincide with the requirements.

This approach to learning-based testing was evaluated on several case studies in [Fen+13] which demonstrated that LBTest successfully detected errors in the software using an incremental learner described in [MS11].

3. Implementation

In this chapter, the implementation of the L* algorithm in an evaluation environment will be described in Section 3.1. Additionally, Section 3.2 will explain the integration of L* Mealy into the LBTest software.

3.1. Evaluation framework for L*

In order to evaluate the properties of Angluin's L* algorithm, the following components were created in software as shown in Figure 3.1

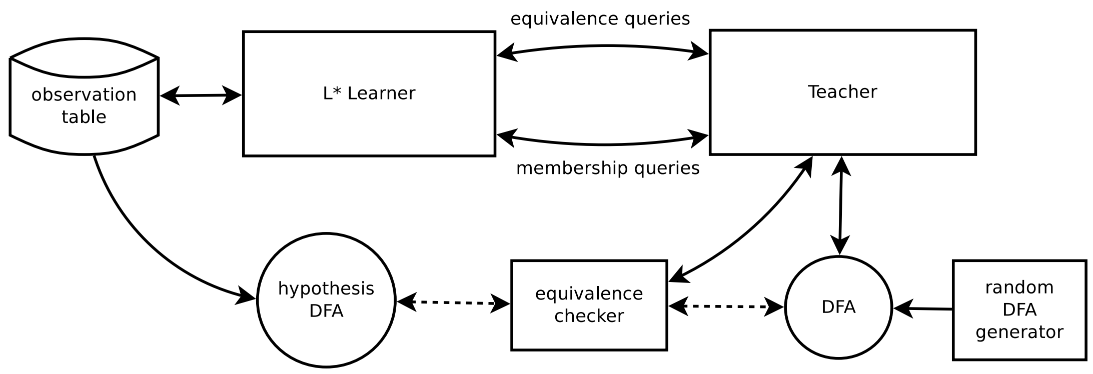


Figure 3.1.: The implemented framework to evaluate the L* algorithm

Framework components

(1) The *L* learner* (see Section 3.1.3) implements the L* algorithm with the previously introduced properties and queries.

(2) The *teacher* (see Section 3.1.2) is a component that has access to the DFA that the learner should understand and that answers the informational needs of the learner without revealing the actual representation.

(3) An *equivalence checker* compares the learner's hypothesis against the, to the learner unknown, DFA.

(4) A *random DFA generator* (see Section 3.1.1) generates DFAs in a random manner. This component only serves as a utility for the evaluation.

3.1.1. Random DFA generator

A random DFA generator was constructed to generate fully connected DFAs. It takes as inputs the input alphabet Σ, the percentage of accepting states and the total number of states. It uses a straight-forward implementation that randomly finds following states for each symbol in the alphabet and for each state, connects these states, randomly assigns accepting states, makes a random state to the initial state and finally checks whether a fully connected graph was created. If the DFA had unreachable states, these would not have affected the behaviour. Thus, these DFAs would be behaviourally equivalent to an automaton with fewer states which could inadvertently bias the measured state space size. To ensure a fully-connected graph a breadth-first search [SM08] checks whether all states can be reached from the initial state using strings generated from the alphabet. If a graph is not connected, it is discarded and the generator repeatedly tries to find a valid connected DFA.

The generated automata have to be mutually different in order to obtain sound data from the measurements. Consequently, the generator neglects all DFAs that are structurally equal to a previously evaluated DFA within one run. Therefore, each automaton is assigned a signature that textually comprises the transition function δ and the accepting states F. Thereafter, the string hash values can be rapidly compared to detect duplicates. This approach does not exclude behaviourally equivalent DFAs that are not equal because minimizing the generated automata is time-consuming and might not represent software in practice.

3.1.2. The Teacher

The role of the teacher, as described in Section 2.1, is to answer membership and equivalence queries regarding the actual DFA.

1. *Membership queries* $Q_M(x)$ are executed on the underlying automaton. Depending on the final state of the DFA after the symbol sequence is executed, the response is either accept or reject.

2. *Equivalence queries* $Q_E(A')$ trigger an equivalence check between the learned DFA A' and the hidden DFA A. In this case a symmetric difference $c \in L(A) \oplus L(A') \in \Sigma^*$ of the combined DFAs is checked to determine the equivalence as explained in [Nor09]. As the L* algorithm is proven to learn the minimal equivalent DFA (see [Ang87]) the hypothesis of the learner can directly be compared to an initially minimized version of the actual DFA. A breadth-first search algorithm described in [SM08] then traverses the graph to find a path to an accepting state and returns it as soon as it finds one — i.e. it finds the shortest accepting path. If any path — i.e. sequence of input symbols — is found, the learner inferred a wrong hypothesis. Thus, the shortest path — either a false positive example from the actual DFA or a false negative example from the proposed DFA — is returned as a counterexample c for the learner. Since the teacher therefore always returns the shortest counterexample c or *true* if no path was found, the length m of counterexamples grows linearly in the state space size $n = |Q|$ of the DFA and thus $m \sim n$ (i.e. m proportional to n) as shown in [Ber+03]. The total number of equivalence queries is constrained by the worst case number of observation table entries which Angluin [Ang87] proofed to be $(k+1)(n+m(n-1))n$ where $k = |\Sigma|$ is the alphabet size.

   Thus, the L* learner finishes learning in the worst case with a polynomial number in $|\Sigma|$ and $|Q|$ of membership queries:

$$O((|\Sigma| + 1)(|Q| + m(|Q| - 1))|Q|) = O(|\Sigma||Q|^3) \tag{3.1}$$

	ϵ	a
ϵ	0	1
a	0	0
aa	0	0
$\epsilon \cdot b$	0	0
$a \cdot b$	0	1
$aa \cdot a$	1	0
$aa \cdot b$	0	1

Table 3.1.: Example observation table with $\Sigma = \{a, b\}$, $S = \{\epsilon, a\}$, $RED = \{\epsilon, a, aa\}$ and $BLUE = \{b, ab, aaa, aab\}$

3.1.3. The Learner

The learner acts upon a table data structure containing all previous observations, as shown in Figure 3.1, that can be separated into a $BLUE$ and a RED part as suggested in [Ber+03]. The RED part contains all equivalence classes that are supposed to represent states in the generated hypothesis DFA. The $BLUE$ table contains information about all successor states to which a transition from a state in RED could arrive $BLUE = (RED \cdot \Sigma) \setminus RED$. The observation table is two-dimensional with *prefix strings* $p \in RED \cup BLUE \subseteq \Sigma^*$ in one dimension and *suffix strings* $s \in S \subseteq \Sigma^*$ in the other dimension. A row in the observation table can be defined as a function $row : \Sigma^* \to \{0,1\}^*$ that maps prefixes to the corresponding rows. Similarly, an entry in the observation table is a function $entry : \Sigma^* \times \Sigma^* \to \{0,1\}$ that uniquely identifies an observation with a prefix and a suffix. Table 3.1 shows an example of an observation table.

(1) As described in 2.1.(a), the *completeness* property is vital to creating a valid DFA because the transition function δ requires a successor state for each state and symbol input. Formally the following statement applies:

$$\forall p_1 \in BLUE : \exists p_2 \in RED : row(p_1) = row(p_2) \tag{3.2}$$

If there is no such $p_2 \in RED$, prefix p_1 is added to RED and all successor states $a \in \Sigma, p_1 \cdot a$ are added to $BLUE$. The observation table depicted in Table 3.1 is not complete because for prefix aaa the following takes effect $\nexists p \in RED : row(p) = row(aaa)$.

(2) As introduced in 2.1.(b), *consistency* is an important property of the observation table as it also drives the learning process. In the implementation it is checked according to [Ber+03] and can be formally defined thus:

$$\exists p_1, p_2 \in RED : row(p_1) = row(p_2) \Rightarrow \forall a \in \Sigma, \forall s \in S : entry(p_1 \cdot a \cdot s) = entry(p_2 \cdot a \cdot s) \tag{3.3}$$

If there is a suffix $s \in S$ and a symbol $a \in \Sigma$ such that the property does not hold, the concatenation $a \cdot s$ is added to the suffixes S. The observation table shown in Table 3.1 is not consistent because $row(a) = row(aa)$ but $entry(a \cdot a \cdot \epsilon) \neq entry(aa \cdot a \cdot \epsilon)$. ϵ denotes the empty string.

Formula 3.3 is equivalent to the congruence property on rows described in Formula 3.4.

$$\exists p_1, p_2 \in RED : row(p_1) = row(p_2) \Rightarrow \forall a \in \Sigma : row(p_1 \cdot a) = row(p_2 \cdot a) \tag{3.4}$$

Using these properties, the learning loop can be abstractly described by the following Algorithm 1.

Algorithm 1 L* learning loop to get a complete and consistent hypothesis h

```
while observations are not consistent or not complete do
    while observations are not complete do
        $x_{incomplete} \leftarrow$ shortest incomplete prefix from $BLUE$
        move $x_{incomplete}$ to $RED$
        add each successor string $x_s = x_{incomplete} \cdot \Sigma$ to $BLUE$ if $x_s \notin BLUE \cup RED$
        perform membership queries for new unknown table entries
    end while
    if observations are not consistent then
        $x_{inconsistent} \leftarrow$ shortest inconsistent suffix from $RED$
        add a new column with $x_{inconsistent}$ to the observation table
        gather new unknown table entries in $RED$ from $BLUE$
        perform membership queries for new unknown table entries in BLUE
    end if
end while
$h \leftarrow$ build hypothesis
```

3.2. Integration of L* Mealy into LBTest

For integration into LBTest, the automaton learner as depicted in Figure 2.2 was implemented using the L* Mealy algorithm previously introduced in Section 2.2. The basic process of the L* Mealy implementation is equivalent to the learning loop of the L* learner depicted in Algorithm 1. Only the *row* and *entry* function (used for instance for checking the observation table properties) as previously described in Section 3.1.3 have to be redefined following the structure of a Mealy automaton specified in Formula 2.2:

$$\begin{aligned} row &: \Sigma^* \rightarrow \Omega^* \\ entry &: \Sigma^* \times \Sigma^* \rightarrow \Omega \end{aligned}$$

The screenshot in Figure A.4 was taken during a testing run with the implemented L* Mealy learner. The LBTest tool measures the number of membership queries, the number of random test cases, the number of test cases by the model checker and the hypothesis size.

4. Evaluation

This chapter makes use of the previously described implementation of L* and L* Mealy to generate data from an abstract learning framework (Section 4.1) and from real industrial software (Section 4.2). All data in this chapter were gathered from executions on a machine with an Intel®Core$^{\text{TM}}$i7-3667U processor with 2.50 GHz clock speed and 4 GB RAM. The operating system was a 64-bit version of Windows 8.1. All components were written in Java with the Java Development Kit 7.

4.1. L*

In this section the generated data from the learning framework presented in Section 3.1 will be shown. First, some properties of the underlying randomly generated automata will be examined in Section 4.1.2. Thereafter, two distinct sets of such automata will be considered in depth. For both of them different statistically significant data was observed from the implemented L* Learner and will be described in order to grasp the behaviour of membership and equivalence queries as introduced in Section 2.1 for different DFAs. Mutual influences of the two query types will be considered.

4.1.1. Simple case study

This section will present a simple case study to clarify the learning process used by L* using queries and reasoning about the contents of the observation table. The DFA, as specified by the graph in Figure 4.1, accepts all strings $s \in \Sigma^*, \Sigma = \{a, b\}$ that contain exactly one b. Thus, the automaton $A_{hidden} = (Q, \delta, q_0, F)$ can be defined using the transitions from the graph for δ, the sets $Q = \{zero, one, trash\}$ and $F = \{one\}$ and the initial state $q_0 = zero$.

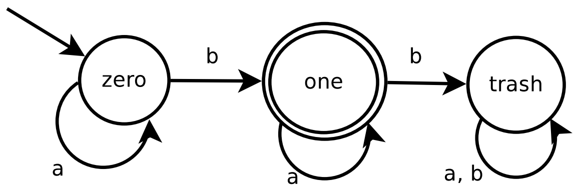


Figure 4.1.: DFA that accepts all strings from the alphabet that contain exactly one 'b'

Learning process

The following sequence shows how the implemented L* algorithm iteratively learns the automaton A_{hidden} from a teacher.

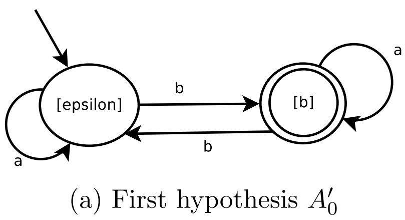
(1) First, the L* learner initializes the observation table according to the current alphabet. The empty string ϵ is added to RED and all symbols $a \in \Sigma$ are added to $BLUE$. The suffixes are initially set to $S = \{\epsilon\}$. Then, all unknown row entries are resolved using membership queries: $Q_M(\epsilon) = 0, Q_M(a) = 0, Q_M(b) = 1$. The initial state of the observation table is shown in Table 4.1a.

(2) Following the learning loop presented in Algorithm 1, each iteration requires to check the consistency and the completeness properties defined in Definition 3.2. The prefix b breaks the formula for completeness 3.3 as there is no equal row in RED to the row that b references. Thus, it has to be moved to RED and successive prefixes $\{ba, bb\}$ have to be added to $BLUE$. The newly added observation table entries have to be resolved using membership queries. The current state of the observation table is depicted in Table 4.1b.

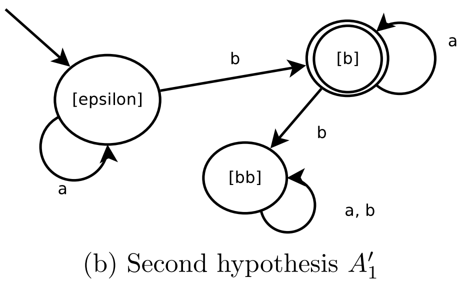
(3) The observation table is now complete and consistent. Thus the L* learner builds a hypothesis A'_0 shown in Figure 4.2a from the observation table and performs an equivalence query $Q_E(A') = bbb$. The counterexample implies that A'_0 was not equivalent to A_{hidden}. Thus, it has to be incorporated into the observation table. Therefore, all prefixes and the counterexample are added or moved to RED and all successive prefixes are added to $BLUE$ if they are not in $BLUE \cup RED$. After the membership queries were performed, the state of the observation table is shown in Table 4.1c.

(4) The learning loop is iterated again because the learning process has not finished with the incorrect hypothesis A'_0. Although the observation table is complete, the suffix b contradicts the consistency property from Formula 3.2. Thus, it is added to the suffixes S and the entries of the observation table for the new column have to be resolved. All entries in RED can be resolved from known values in $BLUE$, as Table 4.1c shows. The unknown entries in $BLUE$ have to be resolved using membership queries. The current state of the observation table is shown in Table 4.1d.

(5) The observation table is complete and consistent so another hypothesis A'_1, as depicted in Figure 4.2b, is built. The equivalence query $Q_E(A'_1) = \emptyset$ reveals that no counterexamples to the correctness of the hypothesis could be found — i.e. $A_{hidden} \equiv A'_1$. In total, the L* learner needed 14 membership queries and 2 equivalence queries to learn the DFA A_{hidden}.



(a) First hypothesis A'_0



(b) Second hypothesis A'_1

Figure 4.2.: Hypotheses from the L* Learner that were generated during the learning process of A_{hidden} as specified by the graph in Figure 4.1

	ϵ
ϵ	0
a	0
b	1

(a) Initial layout of the observation table

	ϵ
ϵ	0
b	1
a	0
ba	1
bb	0

(b) Observation table after the prefix 'b' was moved to *RED*

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

(c) Observation table after the equivalence query returned a counterexample

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

(d) Observation table after the suffix 'b' was added to S

Table 4.1.: Observation table updates to correctly learn the DFA from the graph in 4.1

4.1.2. DFA properties

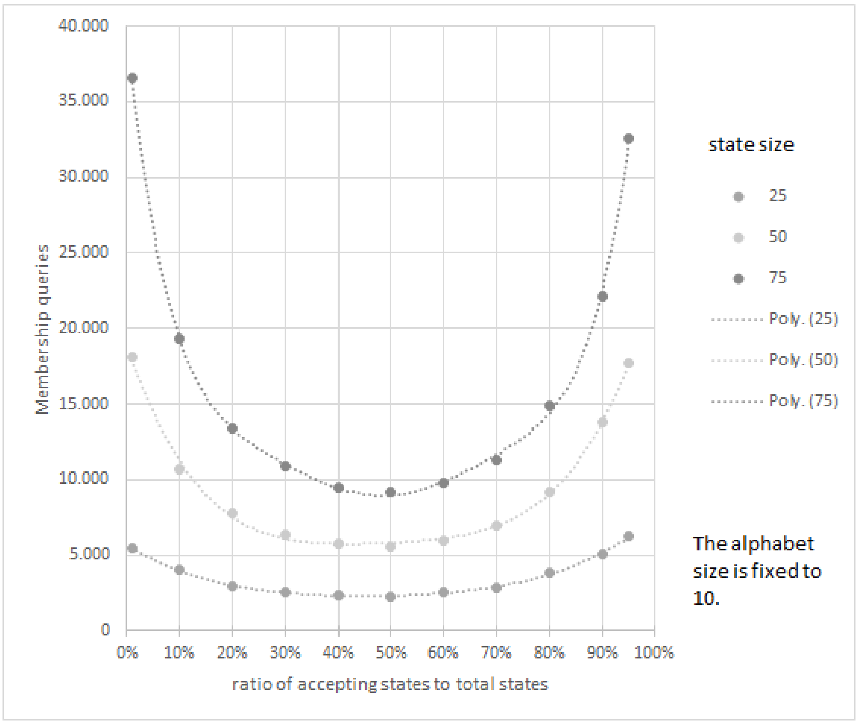
A DFA $A = (Q, \delta, q_0, F)$ as previously defined, has different properties which can impact the behaviour of L^*. In particular, the state space size $|Q|$ of the randomly generated DFAs that L^* is supposed to learn, the alphabet size $|\Sigma|$ and the ratio of accepting states to total states $r := \frac{|F|}{|Q|}$ can be varied individually to influence the difficulty of learning. The transition function δ is set up to always have a connected graph — i.e. it does not contain disconnected states that cannot be reached from the initial state q_0.

The ratio of accepting states $0\% < r < 100\%$ was measured using fixed values for alphabet size and state space size and only varying the number of accepting states $q_{\text{acc}} \in F \subset Q$. The displayed values are average values calculated from 500 learning outcomes on different and randomly generated DFAs with the constraint that $|\Sigma| = 10$ throughout all program runs. Figure 4.3a presents the average membership queries and Figure 4.3b shows how the learner adopts the equivalence queries.

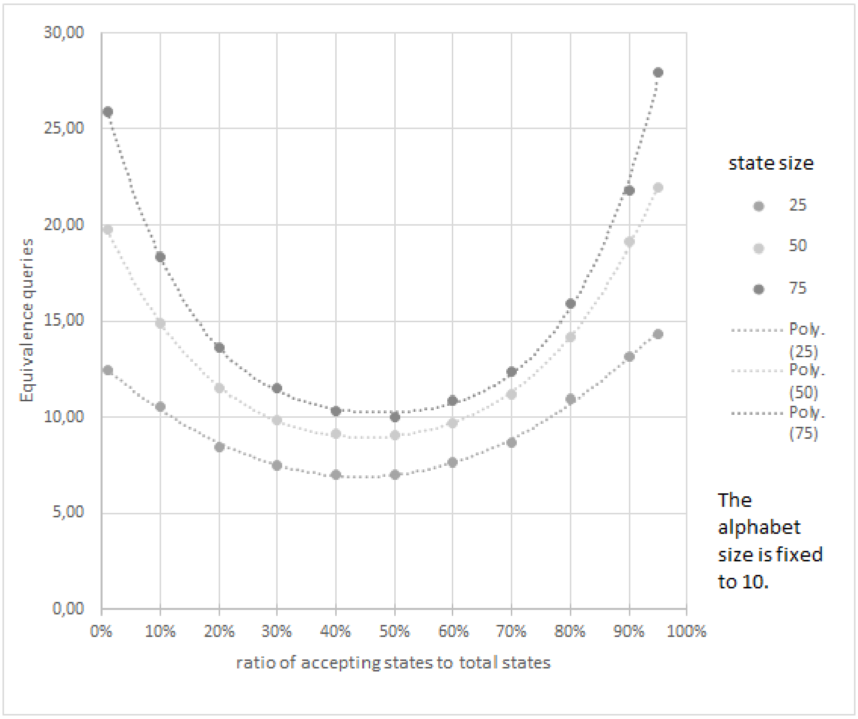
The figures both display a local minimum for the number of queries needed to learn the underlying DFAs with approximately 50% ratio of accepting states. Due to the comparatively small number of computationally expensive membership queries, Section 4.1.3 refers to automata with $r \approx 50\%$ as "*easy-to-learn*". In contrast, Section 4.1.4 deals with the situation of *hard-to-learn* DFAs with an accepting state ratio of $r \approx 1\%$ which is close to the global maximum.

4.1.3. Easy-to-learn automata

In this section, the behaviour of the L* learner will empirically be analysed with the fixed accepting ratio of $r = 50\%$ for the DFAs (which the previous section referred to as easy-to-learn automata). To get a general overview over the learner's behaviour, Figure 4.4 depicts the number of membership queries as a function of the alphabet size $|\Sigma|$ and the state space size $|Q|$ of the learned DFAs. The data was gathered from the previously



(a) Membership queries



(b) Equivalence queries

Figure 4.3.: Behaviour of learning queries for different ratios r of accepting states to total states

	$\lvert\Sigma\rvert \leq 50$
$\lvert Q\rvert \leq 30$	100
$30 < \lvert Q\rvert \leq 75$	500
$75 < \lvert Q\rvert \leq 100$	1000
$100 < \lvert Q\rvert$	100

Table 4.2.: Number of test runs used to evaluate the L* learner with DFAs of different state and alphabet sizes

described learning framework running the learner on similar adjustments for the randomly generated automata. For each value the generator constructed multiple DFAs with the same $|\Sigma|$, $|Q|$ and r but different transition functions δ and different sets of accepting states F. After the L* algorithm successfully learned the DFAs, the arithmetical average of all values was used to generate the charts. The parameters were constrained to $|\Sigma| \leq 50$ and $|Q| \leq 100$ for most of the executions. Within this range, Table 4.2 shows the number of test runs for each evaluation. For bigger state sets, a higher number of executions reduce the impact of random deviations from the mean value. For DFAs with $|Q| > 100$, fewer runs were performed because learning these is very time-consuming. For those charts that contain trend lines, the corresponding approximated functions and the values[1] of R^2 are included.

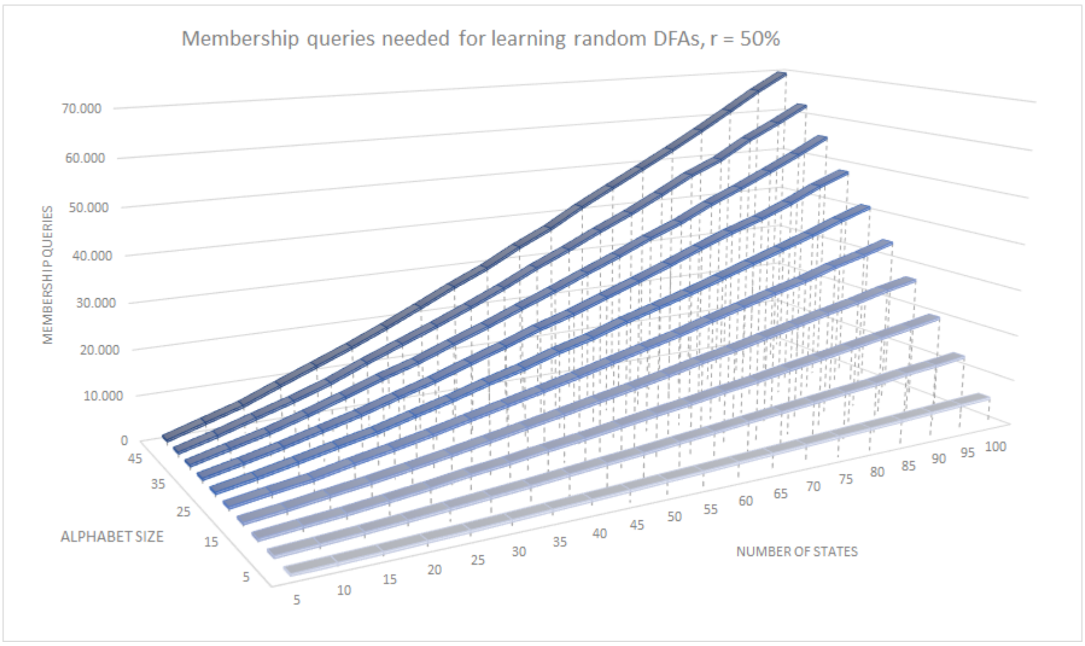


Figure 4.4.: Average membership queries to correctly learn DFAs of different alphabet and state space sizes with $r = 50\%$

While the data for the membership queries was being collected, the number of equivalence queries needed by L* could be measured at the same time. Thus, the method and the parameters remain as for the previously explained chart in Figure 4.4. The following Figure 4.5 shows the number of equivalence queries as a function of $|Q|$ for the test data.

To get a feeling for the behaviour of the previously presented graph in Figure 4.6 showing

[1]The coefficient of determination $0 \leq R^2 \leq 1$ indicates how well the measured data points correlate with a statistical model (here the regression function). $R^2 = 1$ means that the regression function perfectly describes the observed data and lower values imply a less suited approximation. Formally it can be denoted thus: $R^2 := 1 - \frac{s_{res}}{s_{tot}}$, where s_{res} is the residual sum of squares and s_{tot} is the total sum of squares.

how the number of equivalence queries relates to the state space size, higher values for $|Q|$ were used for learning. The procedure remaining the same as before, the average values from 100 test runs were acquired for a greater steps between the measurements of up to $|Q| = 300$.

The chart in Figure 4.7 shows another aspect of L* learning which is the number of equivalence queries as a function of the alphabet size $|\Sigma|$.

To describe the data in more detail, Figure 4.8 shows the same data as the initial Figure 4.4 — i.e. it displays the exact same data so the setup and the constraints are the same. Additionally, suitable trend lines were added to conveniently grasp the overall behaviour of the number of membership queries as a function of $|Q|$.

Two data sets were extracted to acquire data for the individual behaviour of the number of membership queries as a function of the states $|Q|$. As before in Figure 4.6, the analysed range was extended to gather data of the behaviour for higher values of $|Q|$. Figure 4.9a shows the results for $|\Sigma| = 10$ and Figure 4.9b for $|\Sigma| = 20$. To observe the number of membership queries as a function of the alphabet size, the data from Figure 4.5 was narrowed down to focus on the trends of $|\Sigma|$ in Figure 4.10. Finally, the previously gathered data on the number of equivalence queries and membership queries from the test runs is combined in Figure 4.11 as a function of the state space size and in Figure 4.12 as a function of the alphabet size to grasp how many consecutive membership queries it takes in average until the L* learner performs an equivalence query.

4.1.4. Hard-to-learn automata

As in the previous section, different DFAs were randomly generated to evaluate the L* Learner using the implemented learning framework described in Section 3.1. The empirical methods were the same but the ratio r of the number of accepting states $|F|$ to the number of total states $|Q|$ was set to $r = 1\%$. Thus, following the terminology from Section 4.1.2 these DFAs are hard-to-learn — i.e. they require more queries by the learner than automata with equal state and alphabet size but a balanced ratio as depicted in Figure 4.3.

First, the data on membership queries is shown in both dimensions, similar to the previous Section 4.1.3, the number of states and the alphabet size. As before, the learning runs were executed repeatedly on similarly formed DFAs. All depicted data in this section is the average result of 100 single learning outcomes. To understand the behaviour of the number of equivalence queries the charts in Figure 4.14 and Figure 4.16 show the correlation to $|Q|$ and $|\Sigma|$. A broader scale of the evaluation parameters can be seen in Figure 4.15 and both examples from Figure 4.17.

In Figure 4.18 and Figure 4.19 the three-dimensional data from Figure 4.13 is broken down. Eventually, the collective data on the number of membership queries and equivalence queries is correlated in Figure 4.20 and Figure 4.21 as functions of the state space size and the alphabet size.

4.2. L* Mealy

To evaluate the performance of the L* adaptation L* Mealy in practice, two embedded software applications from industry were tested with LBTest. First, a smaller application called *Cruise Controller* 4.2.1 and then the bigger application *Break-By-Wire* 4.2.2 was evaluated. Both examples are thoroughly described by Feng et al. [Fen+13] and could be tested in LBTest with the algorithmic implementation of L* Mealy by courtesy of Volvo AB.

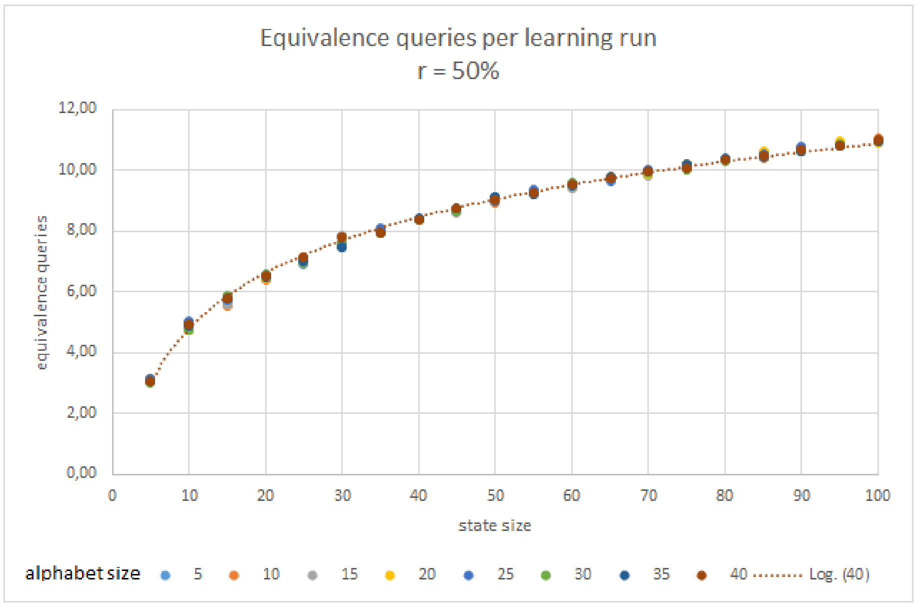


Figure 4.5.: Average equivalence queries to correctly learn DFAs of different state space sizes with $r = 50\%$

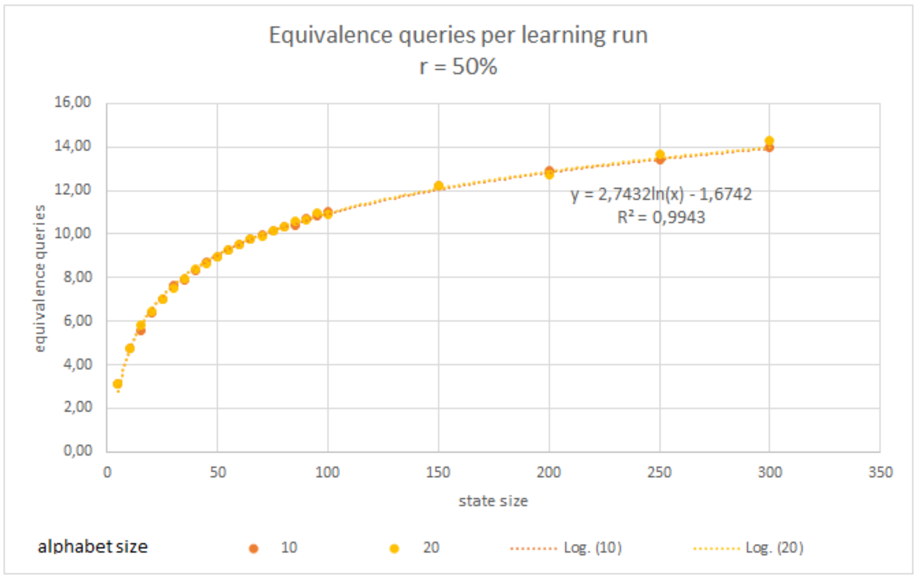


Figure 4.6.: Average equivalence queries to correctly learn DFAs of different state space sizes $|Q|$ for selected alphabet sizes $|\Sigma|$ on a broader scale than Figure 4.5 with $r = 50\%$

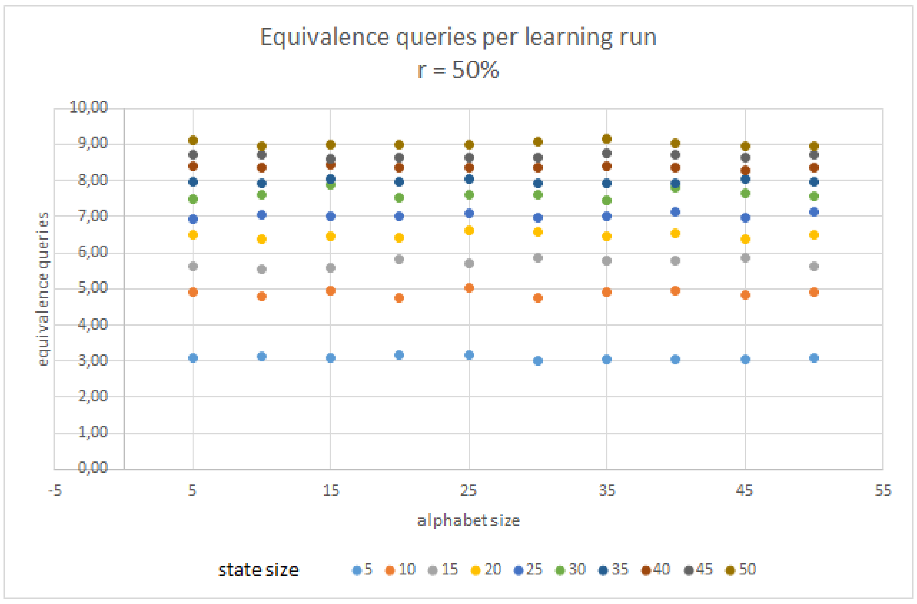


Figure 4.7.: Average equivalence queries to correctly learn DFAs of different alphabet sizes $|\Sigma|$ with $r = 50\%$

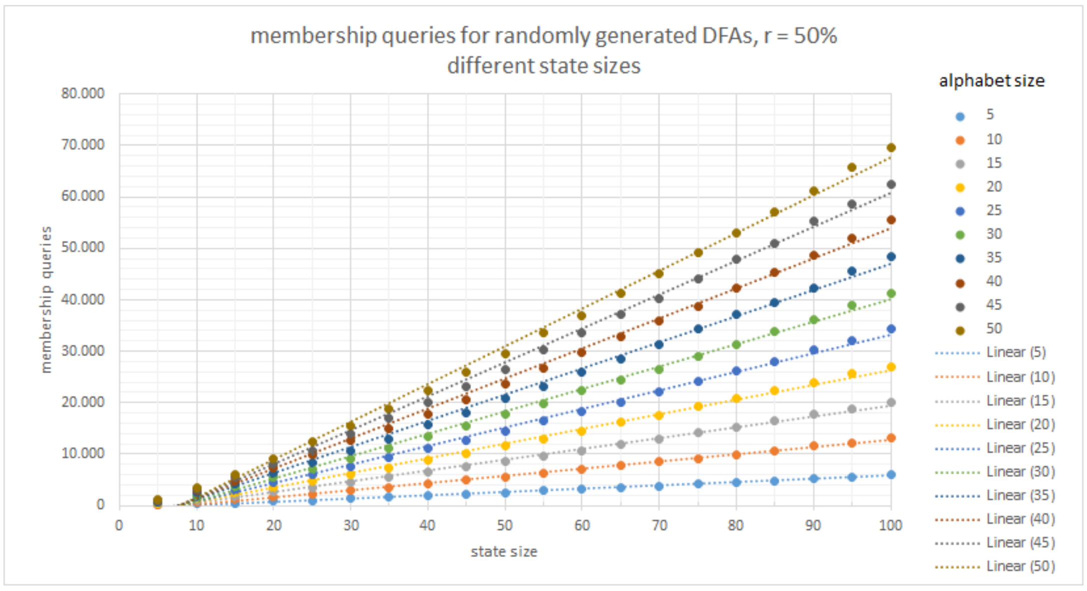
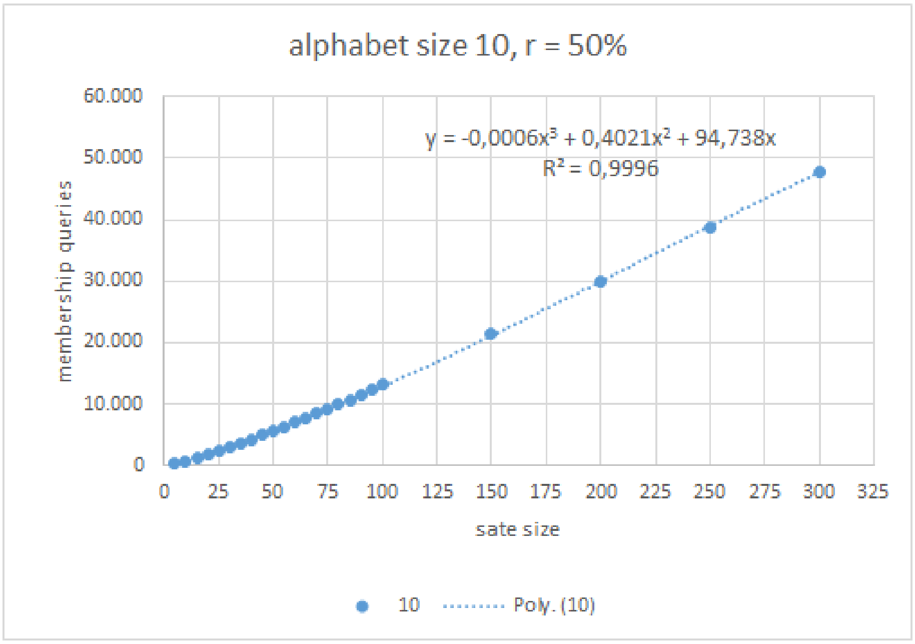
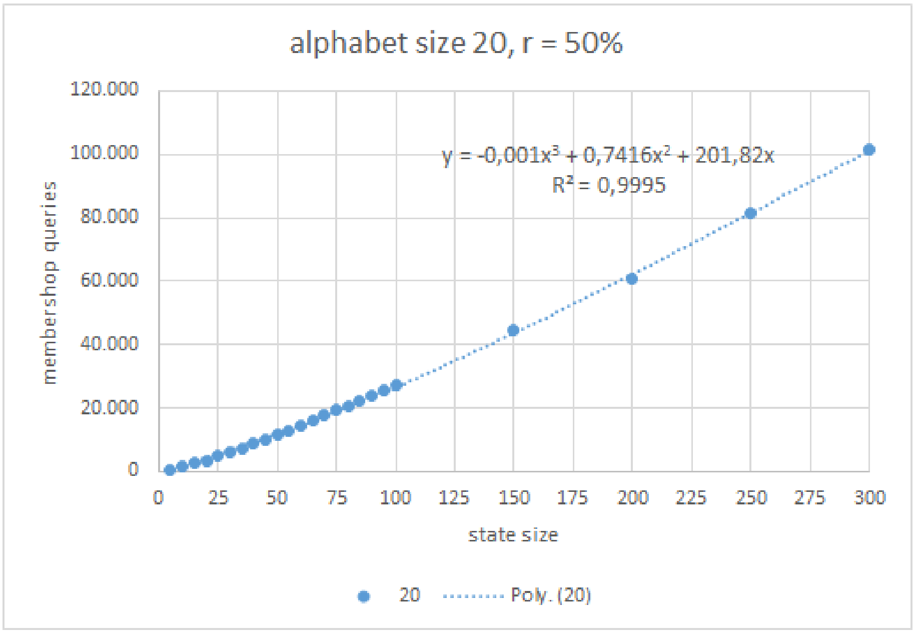


Figure 4.8.: Average membership queries to correctly learn DFAs of different state space sizes with $r = 50\%$



(a) $|\Sigma| = 10$



(b) $|\Sigma| = 20$

Figure 4.9.: Single graphs of the average membership queries to correctly learn DFAs of different state space sizes for selected alphabet sizes with $r = 50\%$

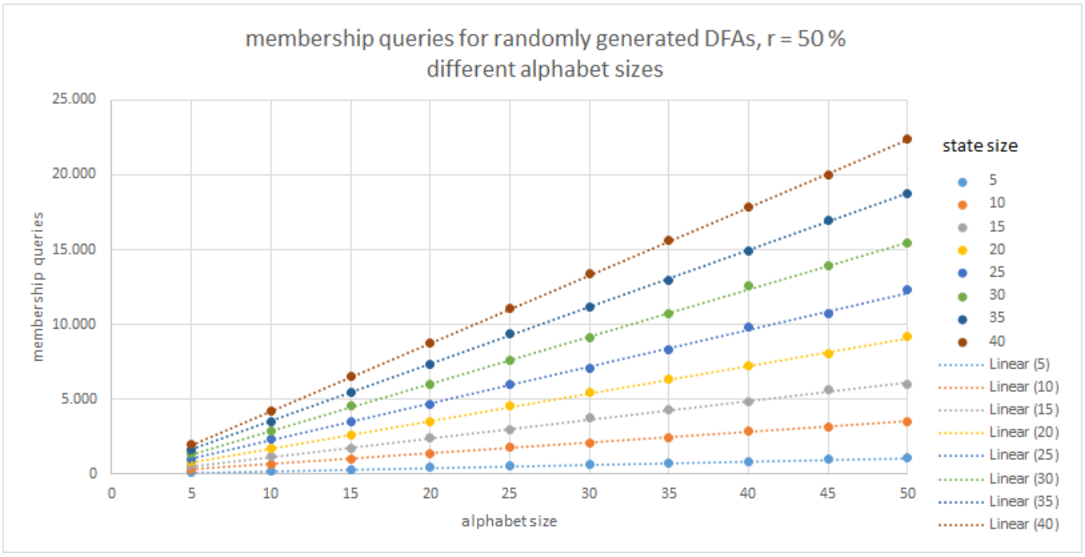


Figure 4.10.: Average membership queries to correctly learn DFAs of different alphabet sizes with $r = 50\%$

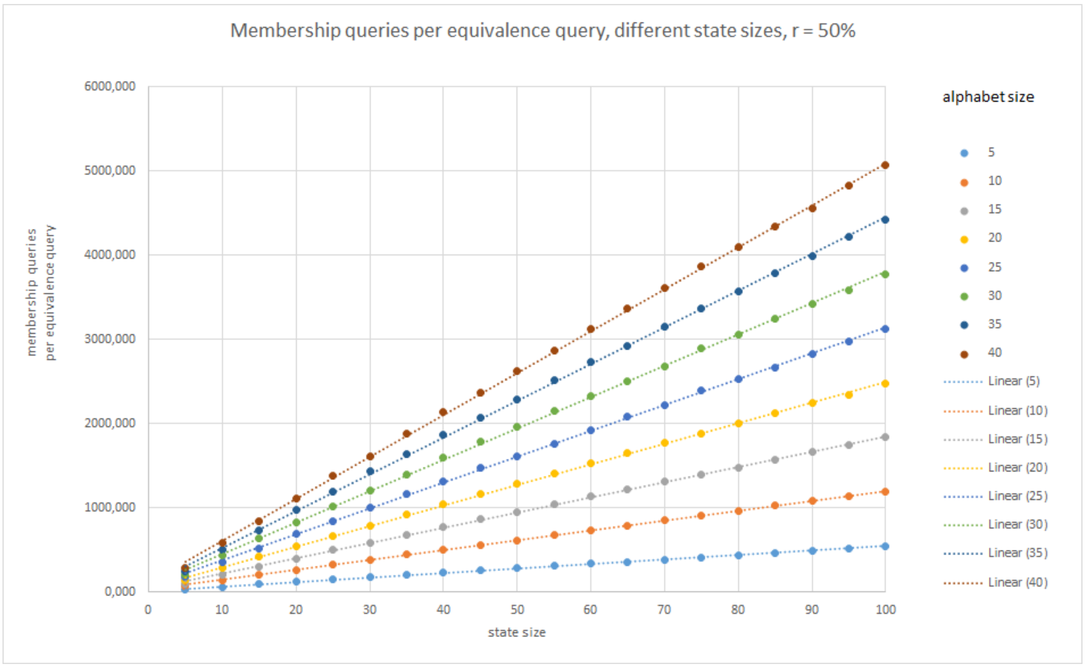


Figure 4.11.: Average membership queries the learner asks before it performs an equivalence query with $r = 50\%$

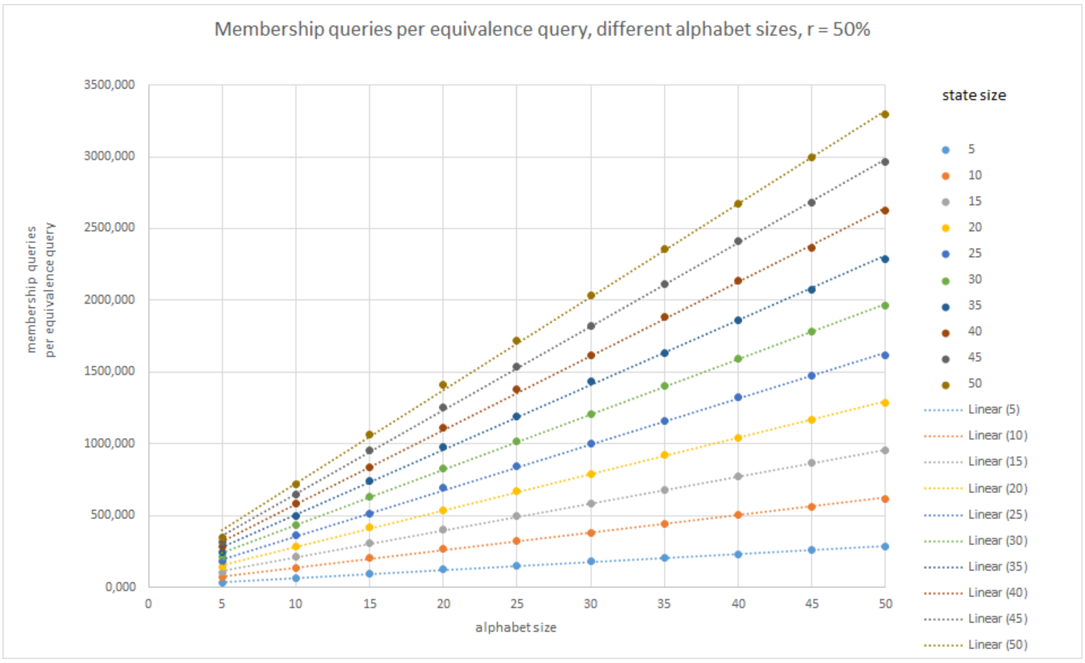


Figure 4.12.: Average membership queries the learner asks before it performs an equivalence query with $r = 50\%$

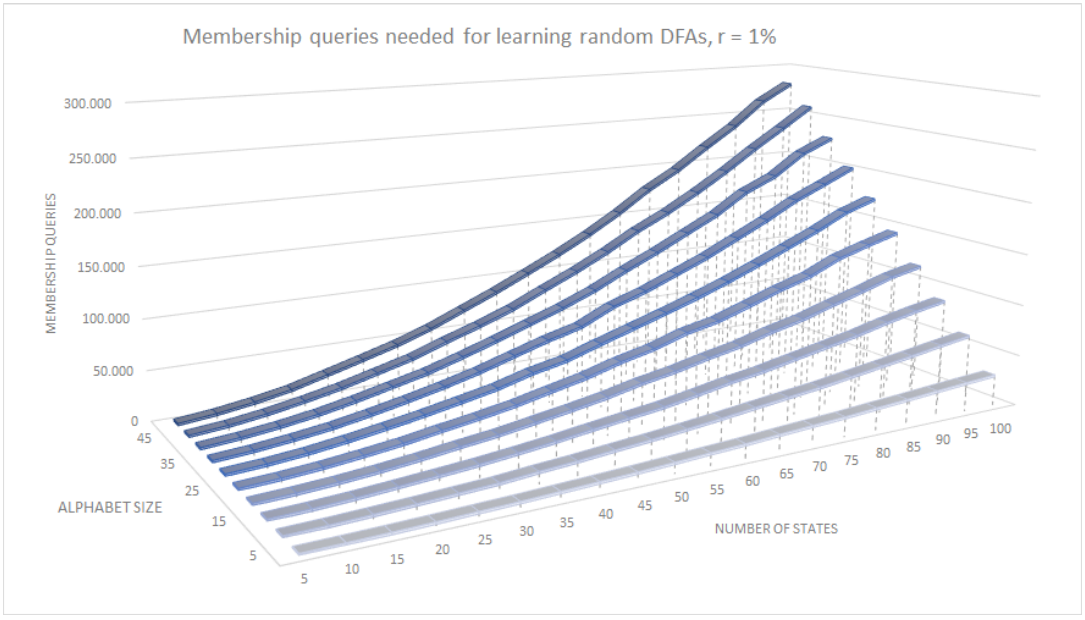


Figure 4.13.: Average membership queries to correctly learn DFAs of different alphabet and state space sizes with $r = 1\%$

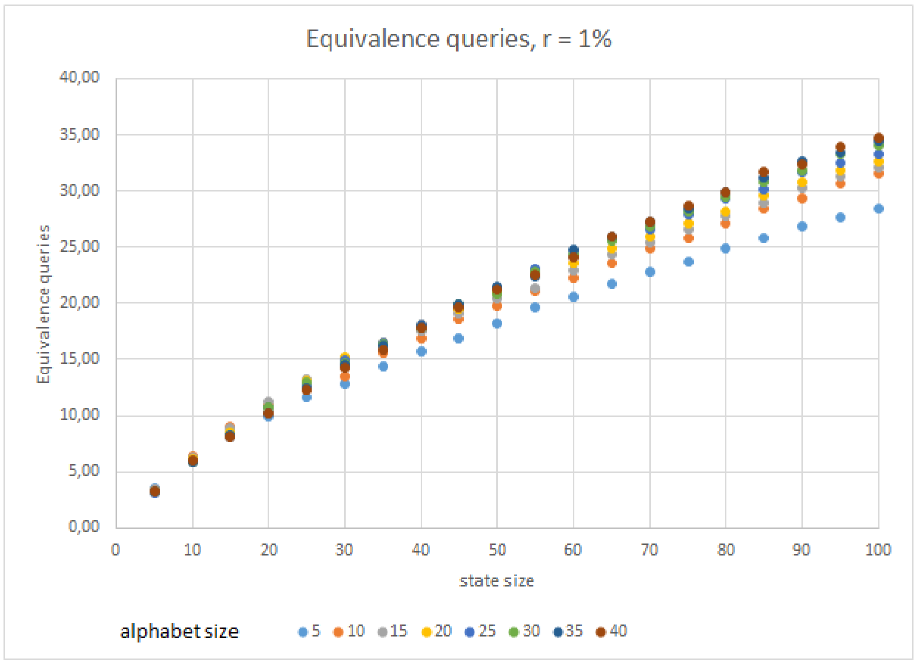


Figure 4.14.: Average equivalence queries to correctly learn DFAs of different state space sizes with $r = 1\%$

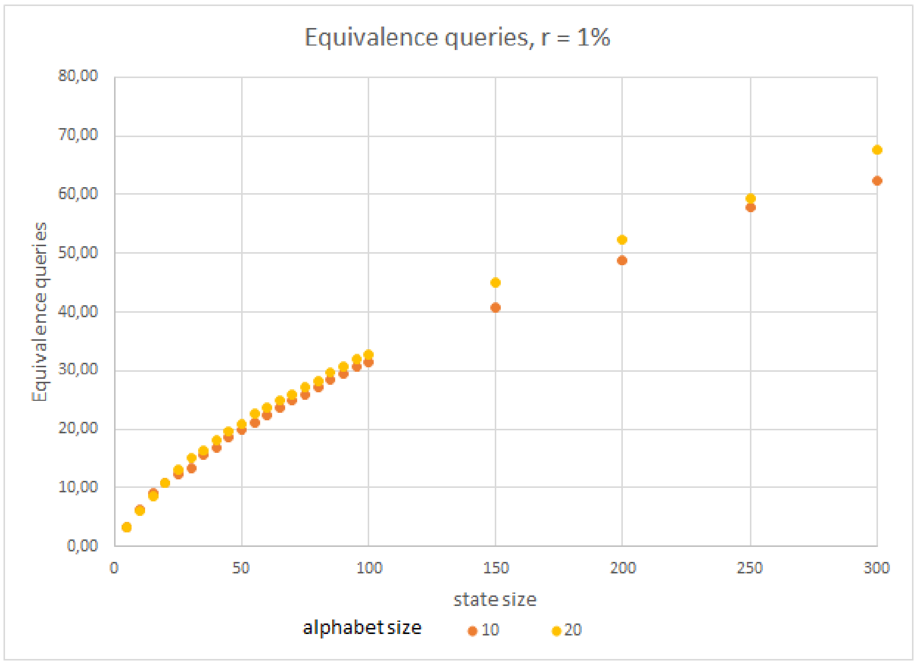


Figure 4.15.: Average equivalence queries to correctly learn DFAs of different state space sizes $|Q|$ for selected alphabet sizes $|\Sigma|$ on a broader scale than Figure 4.14 with $r = 1\%$

4.2.1. Cruise Controller

Cruise control is a system that automatically maintains and regulates the speed of a vehicle. The software requirement against which LBTest tested the application is specified in Table 4.3.
The L* Mealy algorithm successfully learned the *Cruise Controller* (CC) software with only one equivalence query as the screenshot in Figure A.1 shows. The generated correct hypothesis with the structure of a Mealy automaton can be seen in Figure A.3. The LBTest tool yielded the data in Table 4.4 for the test run. In comparison, the same SUT was tested with an incremental learner [MS11], [NP10] as depicted in Figure A.2. The results thereof are collected in Table 4.5.

	description and PLTL
Requirement 1	*If the cruise mode is enabled in the vehicle and the gas is pressed, the mode should be disengaged.*
	$G(mode = cruise \wedge input = gas \rightarrow X(mode = disengaged))$

Table 4.3.: Specification of the the requirement of the *Cruise Controller* application in plain English and propositional linear temporal logic (PLTL) as denoted in the received software specification and described by Meinke and Sindhu [MS13]

#EQ	#MQ	hypothesis size	# random tests	# model checker tests	time (in sec)
1	41	8	0	0	0

Table 4.4.: Results of LBTest testing the *Cruise Controller* using the integrated L* Mealy learner for Requirement 1

#EQ	#MQ	hypothesis size	# random tests	# model checker tests	time (in sec)
1	5	3	0	0	0
2	10	5	0	0	0.24
3	15	5	0	0	0.41
4	20	6	0	0	0.50
5	25	7	0	1	0.58
6	34	8	0	2	0.67
7	39	8	0	2	0.78
8	44	8	0	2	0.87

Table 4.5.: Results of LBTest testing the *Cruise Controller* using an incremental learner for Requirement 1

4.2.2. Break-By-Wire

The *Break-By-Wire* (BBW) software is a real-time embedded vehicle application developed by Volvo Technology AB. Its purpose is to compute the brake torque at each wheel from sensor data.
As the BBW application uses floating point numbers and the L* Mealy learner can only deal with discrete input and output vectors, a wrapper was written that specifies a limited number of intervals for required parameters. LBTest was set up to test the BBW software with three different wrappers for three specific requirements as specified in Table 4.6. Figure 4.22a and Table 4.7 comprise the data from the LBTest tool for the first requirement, Figure 4.22b and Table 4.8 for the second and Figure 4.22c and Table 4.9 for the third. Equally, Figure 4.23a, Figure 4.23b and Figure 4.23c present the results for the number of equivalence queries. The ratio between membership and equivalence queries is depicted in

Figure 4.24. None of the test runs resulted in a fully learned system, though the given data is descriptive for the learning process and indicates the trend of the learner's operations. For requirements 1, 2 and 3 it was possible to track 29, 40 and 20 iterations of the L* Mealy learning loop presented in Algorithm 1.

	description and PLTL
Requirement 1	*If the brake pedal is pressed and the wheel speed (e.g., the front right wheel) is greater than zero, the value of brake torque enforced on the wheel by the corresponding ABS component will eventually be greater than 0.*
	$G(in = brake \rightarrow F(wheel = wheelRotateFR \rightarrow torque = torqueOnFR))$
Requirement 2	*If the brake pedal is pressed and the actual speed of the vehicle is larger than 10 km/h and the slippage sensor shows that a wheel is slipping, this implies that the corresponding brake torque at the wheel should be 0.*
	$G((slip = slipFR \wedge speed = vehicleMove \wedge in = brake) \rightarrow torque \neq torqueOnFR)$
Requirement 3	*If both the brake and gas pedals are pressed, the actual vehicle speed shall be decreased.*
	$G(in = accbrake \rightarrow X(speed = vehicleDecreased))$

Table 4.6.: Specification of the requirements of the *Break-By-Wire* application in plain English and propositional linear temporal logic (PLTL) taken from [Fen+13]

#EQ	#MQ	hypothesis size	# random tests	# model checker tests	time (in sec)
1	101	10	1	0	0.54
2	2826	159	2	1	11.38
3	4709	217	2	2	19.31
4	30275	810	2	3	159.80
learning not finished (manually aborted after 75 minutes)					

Table 4.7.: Results of LBTest testing the *BBW* software using an L* Mealy learner for Requirement 1

#EQ	#MQ	hypothesis size	# random tests	# model checker tests	time (in sec)
1	4	1	0	0	0
2	387	76	1	1	2.52
3	3019	267	1	2	19.36
4	34060	1251	1	3	303.48
5	55390	1660	1	4	572.35
learning not finished (manually aborted after 60 minutes)					

Table 4.8.: Results of LBTest testing the *BBW* software using an L* Mealy learner for Requirement 2

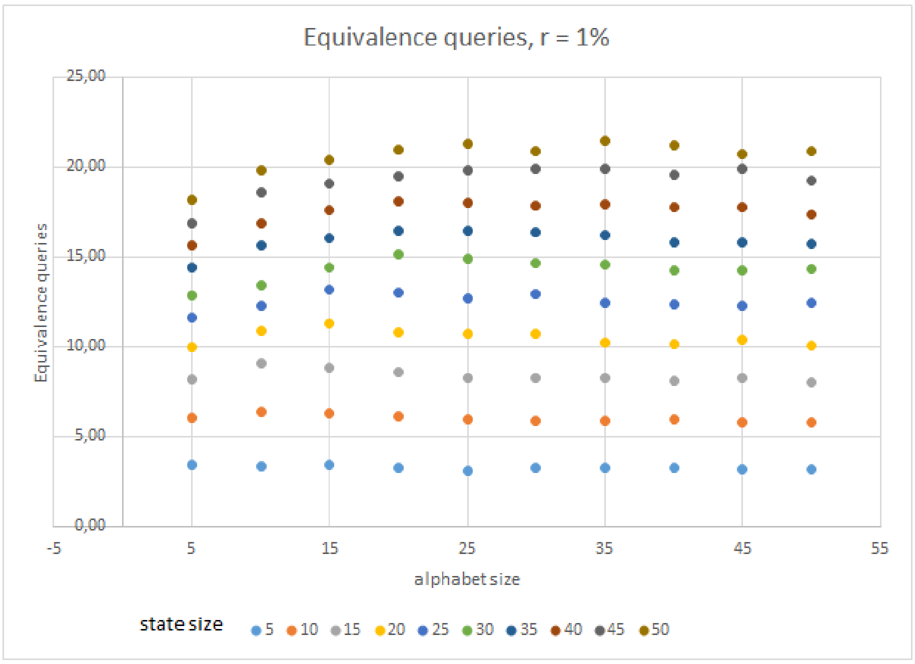
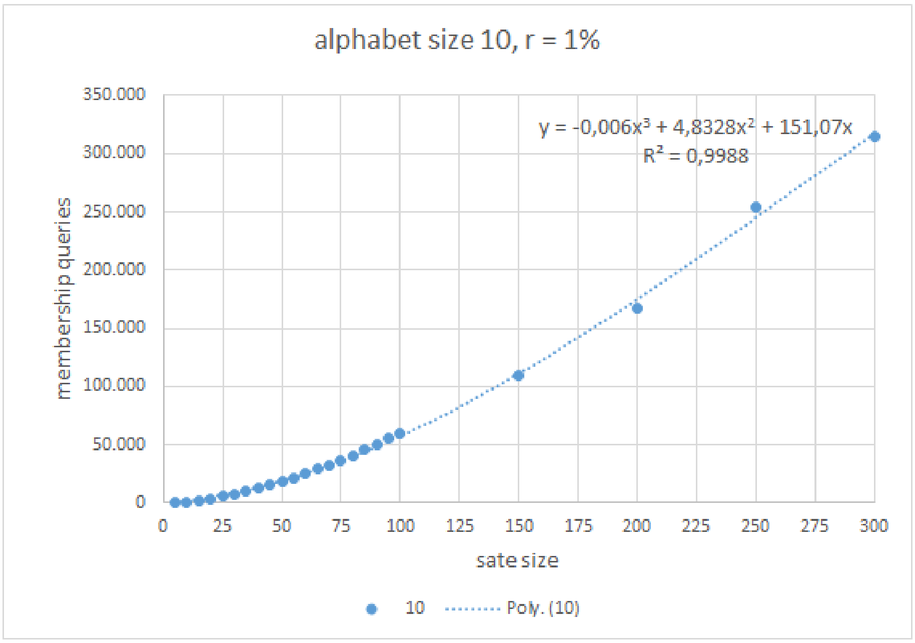


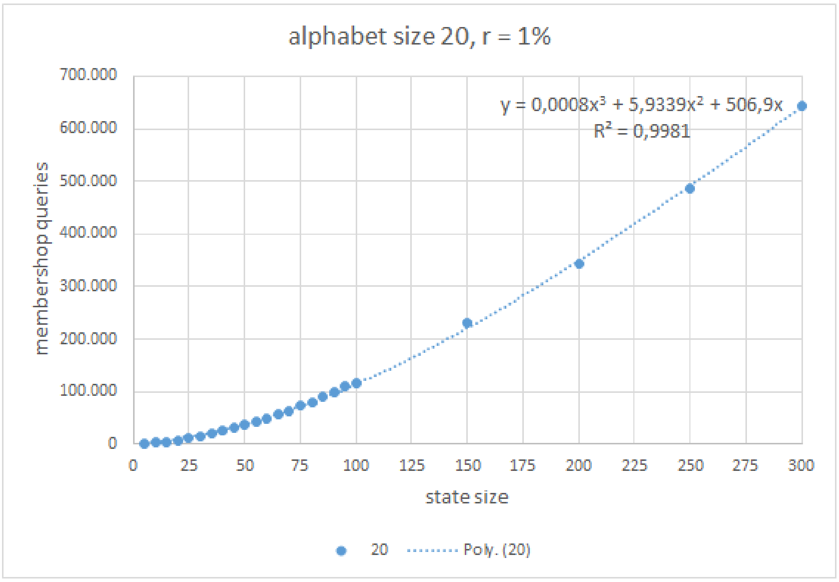
Figure 4.16.: Average equivalence queries to correctly learn DFAs of different alphabet sizes $|\Sigma|$ with $r = 1\%$

#EQ	#MQ	hypothesis size	# random tests	# model checker tests	time (in sec)
1	13	3	0	0	0
2	79	9	1	0	0.52
3	1349	77	2	0	3.23
4	21261	741	5	0	67.89
learning not finished (manually aborted after 60 minutes)					

Table 4.9.: Results of LBTest testing the *BBW* software using an L* Mealy learner for Requirement 3



(a) $|\Sigma| = 10$



(b) $|\Sigma| = 20$

Figure 4.17.: Single graphs of the average membership queries to correctly learn DFAs of different state space sizes for selected alphabet sizes with $r = 1\%$

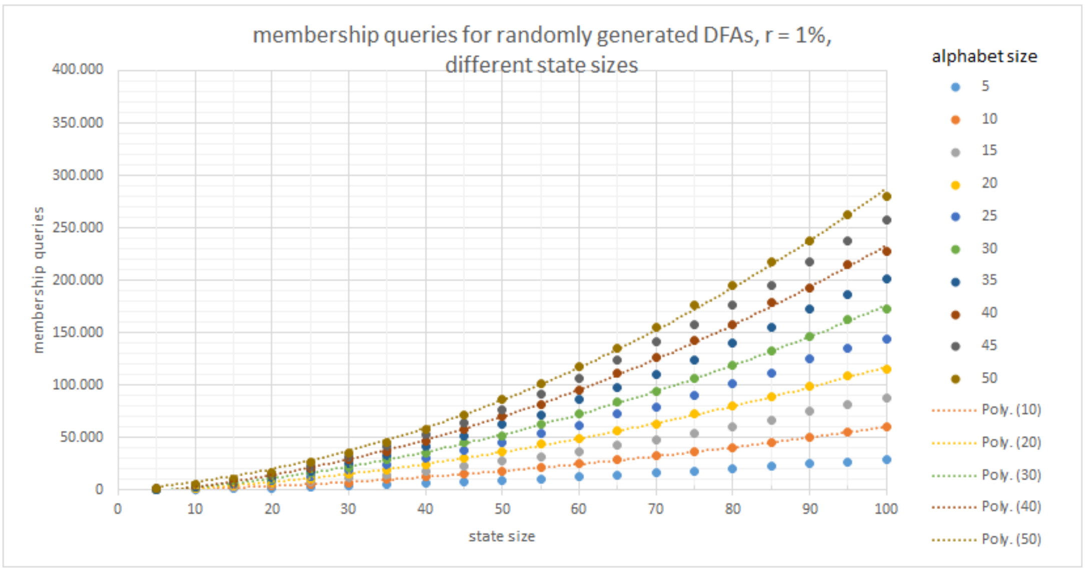


Figure 4.18.: Average membership queries to correctly learn DFAs of different state space sizes with $r = 1\%$

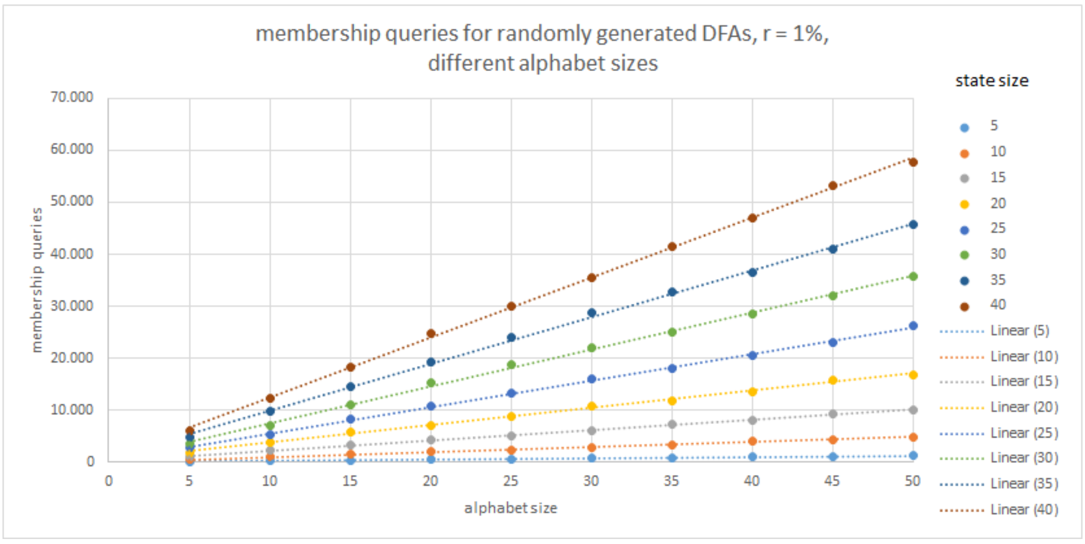


Figure 4.19.: Average membership queries to correctly learn DFAs of different alphabet sizes with $r = 1\%$

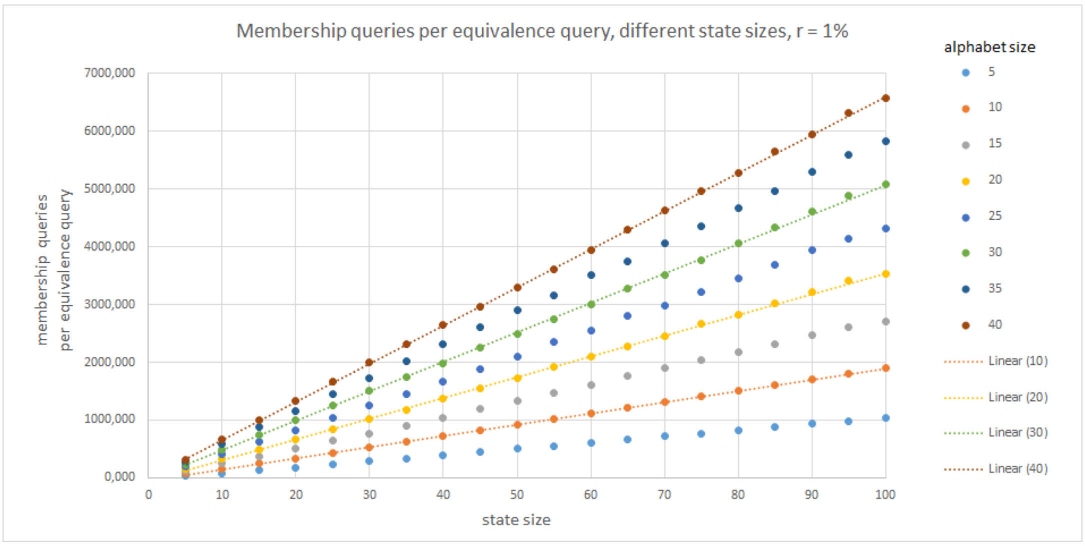


Figure 4.20.: Average membership queries the learner asks before it performs an equivalence query with $r = 1\%$

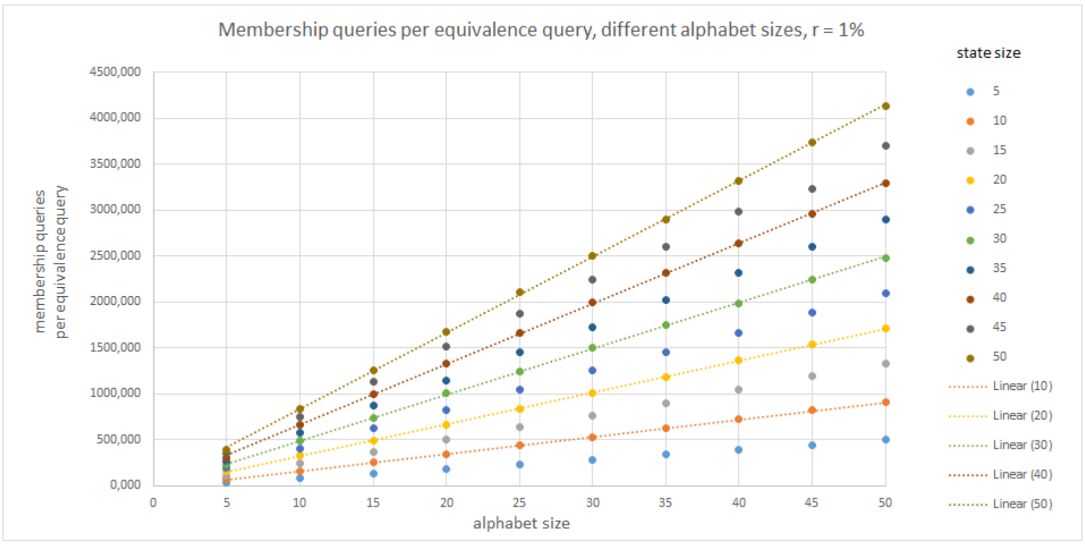
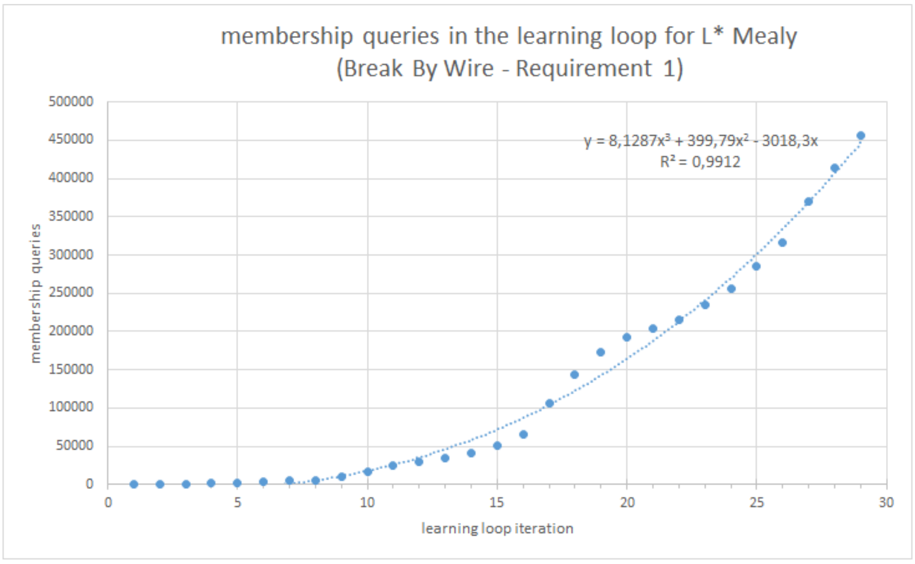
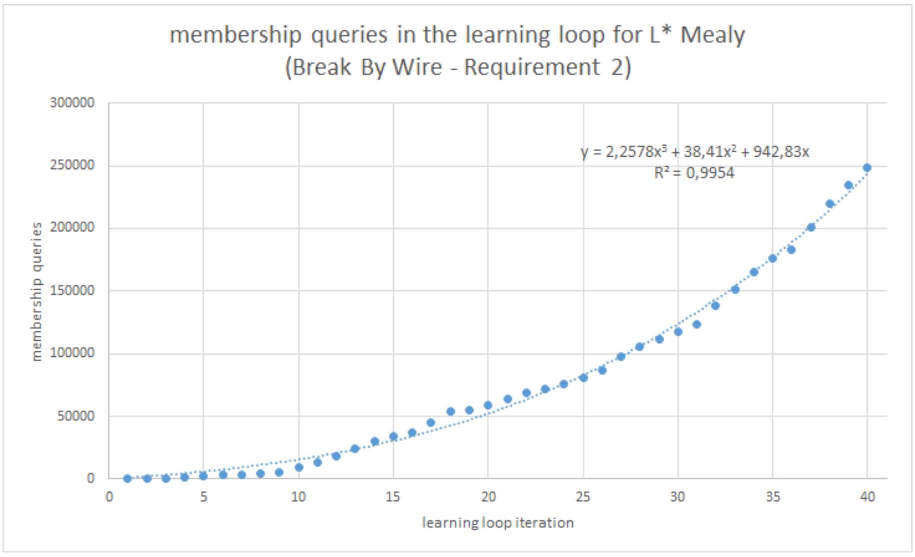


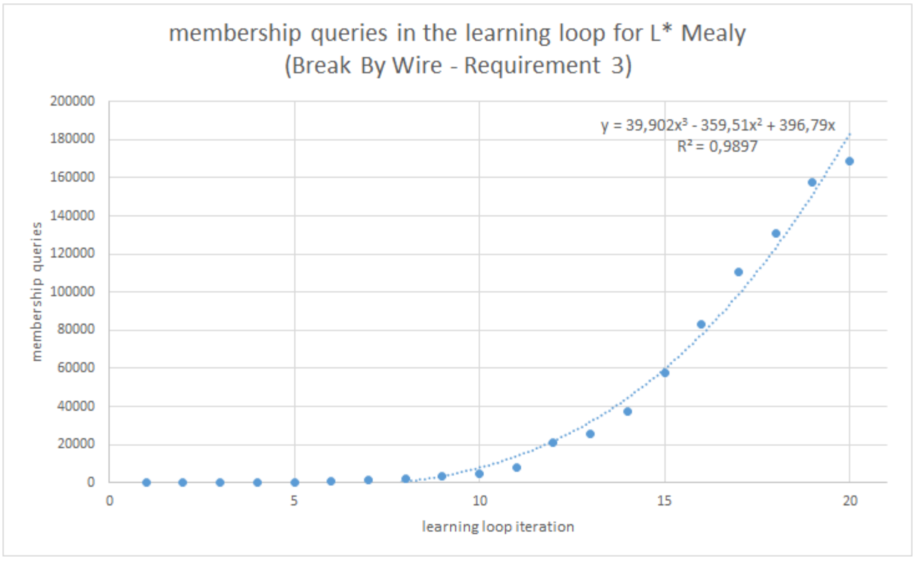
Figure 4.21.: Average membership queries the learner asks before it performs an equivalence query with $r = 1\%$



(a) Requirement 1

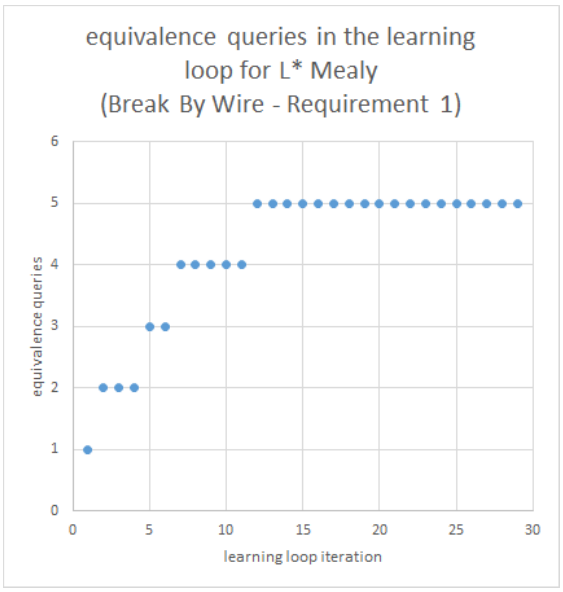


(b) Requirement 2

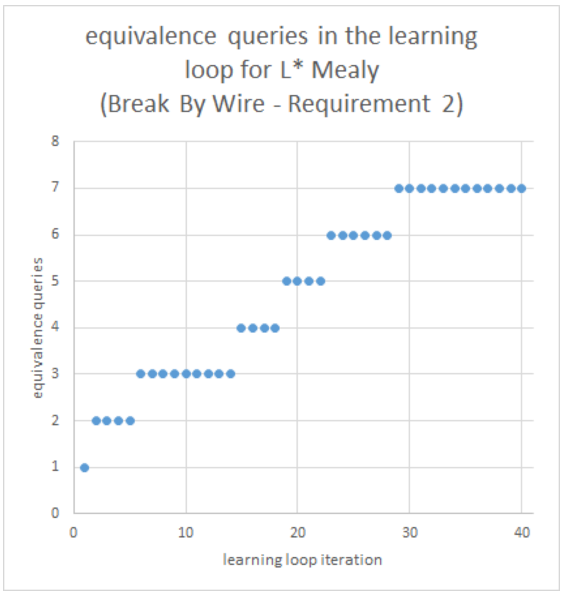


(c) Requirement 3

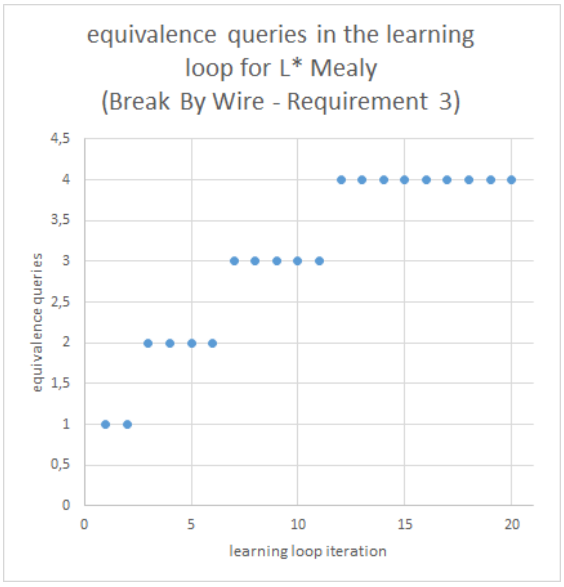
Figure 4.22.: Graphs for the number of membership queries after each iteration of the learning loop in Algorithm 1 for different requirements (they show the data gathered during the executions that were previously described in Table 4.9, Table 4.9 and Table 4.9)



(a) Requirement 1

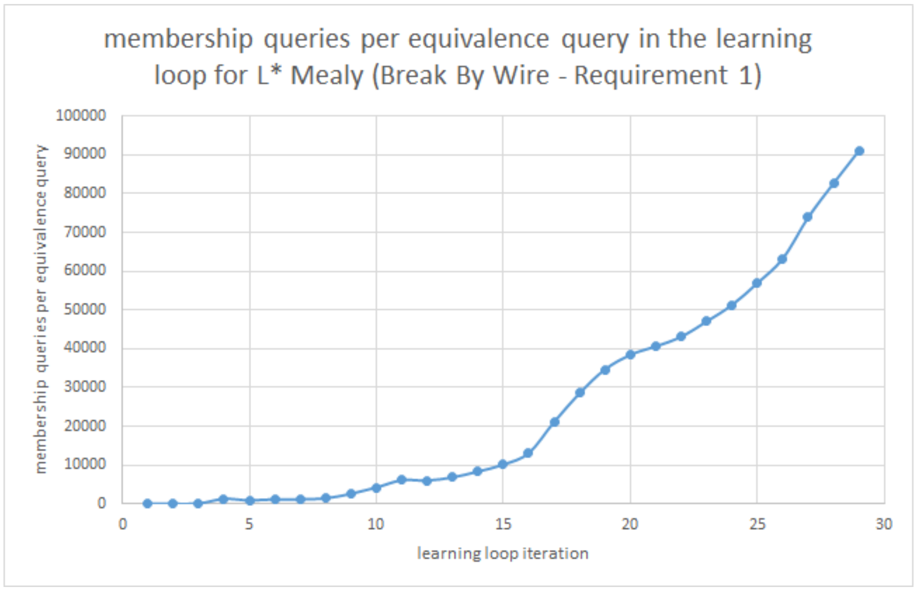


(b) Requirement 2

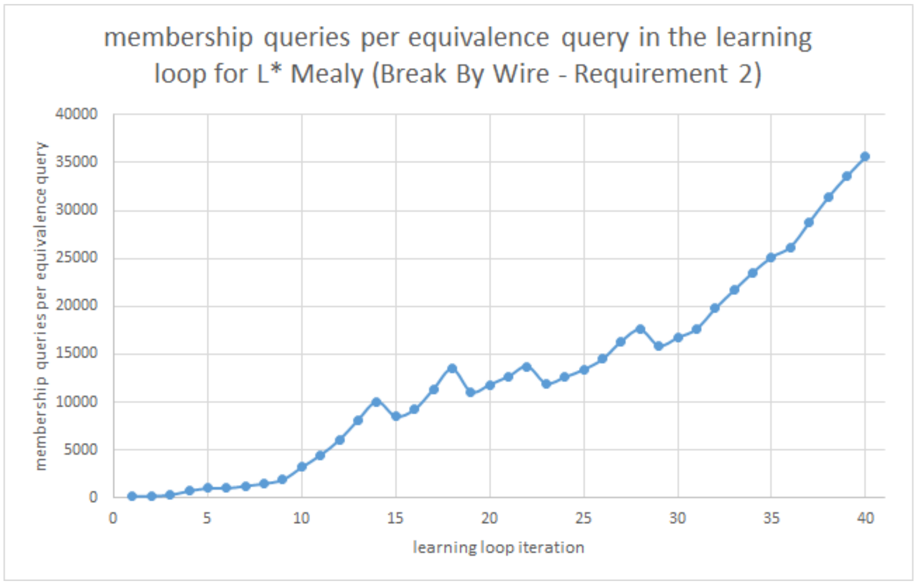


(c) Requirement 3

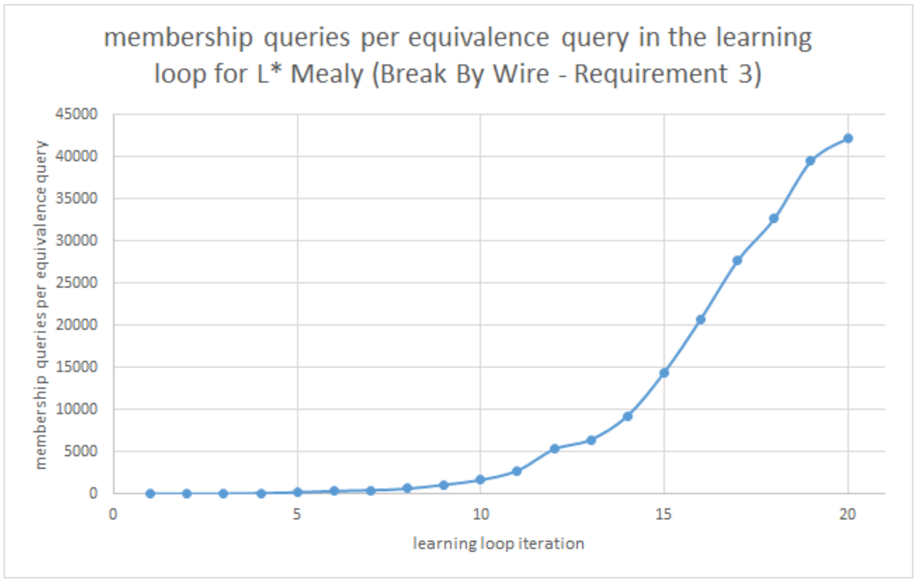
Figure 4.23.: Graphs for the number of equivalence queries after each iteration of the learning loop in Algorithm 1 for different requirements (they show the data gathered during the executions that were previously described in Table 4.9, Table 4.9 and Table 4.9)



(a) Requirement 1



(b) Requirement 2



(c) Requirement 3

Figure 4.24.: Graphs for the number of membership queries per equivalence query after each iteration of the learning loop in Algorithm 1 for different requirements

5. Discussion

Angluin's L* algorithm gives the possibility to learn a hidden DFA, but the teacher has to have knowledge about the internal structure of the automaton. With an extension using the L* Mealy adaptation, the approach could be applied to the more general problem of black-box testing. First, the results of the performance of L* in the implemented framework will be discussed in Section 5.1 and then they will be compared to the performance of L* Mealy using the two case studies in Section 5.2.

5.1. Practical evaluation of Angluin's L* algorithm

The L* results will be discussed in two sections. First, equivalence queries will be considered in Section 5.1.2 and then membership queries in Section 5.1.3. As shown in [Ber+03], the feasibly testable value ranges of state and alphabet sizes in this report can cover a variety of simple and advanced transition systems e.g. communication protocols.

To better describe the empirical behaviour of the L* learner, this section denotes the typically observed number of membership queries as L^*_{MQ} as a function of the alphabet size $|\Sigma|$ and the state space size $|Q|$ of the DFA that should be learned. The mean number of equivalence queries is similarly defined as L^*_{EQ}. It can be formalised thus:

$$L^*_{MQ}, L^*_{EQ} : \mathbb{N} \times \mathbb{N} \to \mathbb{R} \tag{5.1}$$

5.1.1. Correctness of the L* algorithm implementation

The example in Section 4.1.1 shows that Angluin's L* algorithm successfully learns an initially unknown DFA.
The implementation contained assertions on the upper bound of the number of membership and equivalence queries deduced by Angluin [Ang87]. All executions were in full agreement with them. Furthermore, no membership query was performed twice and the observation table was always complete and consistent before building a new hypothesis. Each learning process terminated with a learned DFA that was behaviourally equivalent to the initially unknown DFA. Moreover, the data on the number of membership queries for easy-to-learn DFAs presented in Section 4.1.3 coincides with the evaluation results by Berg et al. [Ber+03]. Both, the trend lines for the number of membership queries as a function of the state size and the overall value range match the data from the implemented learning framework. Furthermore, the findings from several models by Howar [How12] show the same order of magnitude as the presented data.

5.1.2. Equivalence queries

As shown in Equation 3.1, one might expect polynomial growth[1] in the number $q_{MQ}(|Q|) := L^*_{MQ}(|Q|, c) \in O(|Q|^3), c \in \mathbb{N}$ of membership queries that the L* learner needs on average to fully understand the hidden DFA as a function of the state space size $|Q|$. Nevertheless, Figure 4.8 indicates that on randomly generated DFAs with $r \approx 50\%$ a roughly linear behaviour can be observed for different alphabet sizes $|\Sigma|$. If Figure 4.5, Figure 4.14, Figure 4.6 and Figure 4.15 are used as a reference, it is possible to explain how the data relates in this case. The number $q_{EQ}(|Q|) := L^*_{EQ}(|Q|, c), c \in \mathbb{N}$ of equivalence queries seems to grow logarithmically in proportion to $|Q|$. Angluin formally proved in [Ang87] that there will be at most $|Q| - 1$ counterexamples to the consecutive hypothesis of the learner. Thus, including the last correct equivalence query one obtains $q_{EQ}(|Q|) \leq |Q|$. In practice, however, the average results from Figure 4.6 distinctly show that $q_{EQ}(|Q|) \in O(\ln(|Q|))$. Obviously, as the whole, the learner is able to differentiate between the correct equivalence classes much faster than in the worst case scenario in which the teacher can find a counterexample to each new equivalence class.

Apparently, the alphabet size does not contribute to the number of equivalence queries. Figure 4.7 and Figure 4.16 show that for fixed state space sizes $a_{EQ}(|\Sigma|) := L^*_{EQ}(c, |\Sigma|), c \in \mathbb{N}$ stays constant while $|\Sigma|$ grows — i.e. the number of equivalence queries is independent of the alphabet size. This is counter-intuitive because a higher number of symbols entails a more complex representation of the automata and thus, more information to learn for the L* algorithm. This behaviour might be attributed to the properties of the observation table. On the one hand, the learner needs to observe more behaviour. On the other hand, the learner only performs an equivalence query if the observation table is complete and consistent. With more symbols in Σ, this results in more membership queries to accomplish the properties. However, an in-depth investigation is needed, to fully understand this behaviour of a_{EQ}.

5.1.3. Membership queries

The previous section shows that in practice the L* learner performs fewer equivalence queries on the teacher's hidden DFA than expected. With fewer equivalence queries in general, a considerable number of membership queries can be saved. For instance, fewer equivalence queries implies fewer counterexamples and this entails fewer membership queries for partial counterexamples and consequent successive states. Indeed, Figure 4.8 and Figure 4.18 do not show the theoretical worst-case behaviour which, following Angluin's remarks [Ang87], should be $q_{MQ}(|Q|) \in O(|Q|^3)$. Instead, Figure 4.8 indicates that for $r = 50\%$ the number of membership queries grows nearly linearly — $q_{EQ}(|Q|) \sim |Q|$. The inserted polynomial trend lines in Figure 4.9 (see also the approximated cubical functions next to these charts) put the weight on the linear term whereas the quadratic and the cubical term are nearly eliminated. The global minimum of the acceptance ratio, as introduced in Section 4.1.2, of about $r = 50\%$, as depicted in Figure 4.3a, highlights this dramatic difference between the empirically observed average membership queries and the theoretical worst case. If the number of accepting states $|F|$ is approximately equal to the number of rejecting states $|Q \setminus F|$ of the DFA, the learner detects the various equivalence classes faster with the consistency (as introduced in Section 2.1) checks. Assuming that $|F| \ll |Q \setminus F|$, there is a high probability that, given a symbol in any state of a hypothetical acceptor, the automaton ends up in a rejecting state — i.e. the learning progresses only slowly or by counterexamples as the observation table is very likely consistent. For $r = 50\%$, the

[1]The big O notation is used in this report to denote an upper bound of the given mathematical function $f(x)$ and is defined as follows: $f(x) \in O(g(x)) :\Leftrightarrow \exists c \in \mathbb{R}, c > 0 : \exists x_0 \in \mathbb{R}, x_0 > 0 : \forall x \in \mathbb{R}, x > x_0 : |f(x)| < c \cdot |g(x)|$.

opposite takes effect: $|F| \approx |Q \setminus F|$. Thus, there is a higher chance to find an inconsistency in the observation table at any time which drives the learner on without the need for many counterexamples.

At the extreme limit of the ratio realm, such as for $r = 1\%$, the membership queries needed to fully learn the DFAs rise, as previously argued. For instance, Figure 4.18 with $r = 1\%$ shows distinct differences to Figure 4.8 although all other conditions are equal. As a matter of fact, Figure 4.17b and Figure 4.17a show that for $r = 1\%$ the quadratic term prevails in the trend lines. Thus, the depicted data is dominated by polynomial growth.

Independent of r and for $a_{MQ}(|\Sigma|) := L^*_{MQ}(c, |\Sigma|), c \in \mathbb{N}$, both Figure 4.19 and Figure 4.10 support the insight that $a_{MQ}(|\Sigma|) \sim |\Sigma|$ as theoretically predicted by Angluin [Ang87].

Overall, the practical evaluation shows that the learning time of L*, which is determined by q_{EQ} and a_{MQ}, is better than the theoretical worst-case shown in Equation 3.1. For easy-to-learn DFAs (see Section 4.1.3), the empirical results indicate $L^*_{MQ}(|Q|, |\Sigma|) \in O(|\Sigma||Q|)$ and $L^*_{EQ}(|Q|, |\Sigma|) \in O(\ln(|Q|))$ in the average case and for hard-to-learn automata $L^*_{MQ}(|Q|, |\Sigma|) \in O(|\Sigma||Q|^2)$ and $L^*_{EQ}(|Q|, |\Sigma|) \in O(|Q|)$.

5.1.4. Membership queries per equivalence query

This section will discuss the coherence of membership and equivalence queries to better understand the behaviour of the L* learner. As a sequence of membership queries is always followed by an equivalence query, the average number of membership queries between two consecutive equivalence queries (from now on called ρ) is an informative ratio.

Both evaluations, the one for easy-to-learn DFAs and the one for hard-to-learn DFAs, show a linear trend in the growth of ρ as a function of the state and the alphabet size. For $r = 50\%$ Figure 4.11 and Figure 4.12 show this correlation and for $r = 1\%$ Figure 4.20 and Figure 4.12 indicate the same. The linear trends are implications of the behaviour of the queries as discussed in Section 5.1.2 and Section 5.1.3. The numbers of membership queries $a_{MQ_{r=50\%}}(|\Sigma|)$ and $a_{MQ_{r=1\%}}(|\Sigma|)$ grow linearly with the alphabet size $|\Sigma|$. Additionally, the numbers of equivalence queries $a_{EQ_{r=50\%}}$ and $a_{EQ_{r=1\%}}$ stay constant. Thus, $\rho_{r=50\%} \sim |\Sigma|$ and $\rho_{r=1\%} \sim |\Sigma|$. The flattening trends of $q_{EQ_{r=50\%}}$ and $q_{EQ_{r=1\%}}$ as functions of the state space size compensate for the continuously increasing slope of the membership queries.

It can be concluded that equivalence queries are very rare in the learning process in contrast to membership queries. As both return valuable information about the automata to be learned, this imbalance slows down the algorithm with increasing complexity and learning difficulty of the DFAs.

5.2. Case studies with L* Mealy

In this section, the applicability of L* Mealy in the case studies with the Cruise Controller application in Section 5.2.1 and the Break-By-Wire software in Section 5.2.2 will be discussed. Therefore, the observed data from Section 4.2 will be used.

5.2.1. Cruise Controller

The successful learning-based testing run of the Cruise Controller application shows the applicability of complete learning algorithms in the LBTest scenario. As a matter of fact, the L* Mealy algorithm was able to learn the SUT with the first hypothesis as Table 4.4 shows. With slightly less membership queries than the incremental learner (compare Table 4.4 and Table 4.5) and far less learning time, the L* Mealy learner seems to be better suited for LBT than the incremental learner. This is only true because of the small size of the system (see the full Mealy automaton in Figure A.3). Section 5.2.2 discusses the results for a bigger case study. The data from Table 4.4 indicates that CC can be represented by a Mealy automaton with $|\Sigma| = 5, |Q| = 8$. Compared to the L* results in Figure 4.4, the required number of membership queries and equivalence queries was even below the average values for easy-to-learn — i.e. $r = 50\%$ — DFAs of similar complexity. That implies an especially well-suited case study for complete learning.

5.2.2. Break-By-Wire

The BBW case study shows how different communication wrappers for the SUT can mask out certain parts of the system that are not relevant to the currently tested requirement. Thus, the three introduced requirements involve observing the SUT in three different ways. Therefore, particular parts can be individually learned with different emerging models. As opposed to the test of the *Cruise Controller*, the learning-based testing approach seems to backfire for the L* Mealy learner because none of the three executions actually finished the learning process within an hour so they were manually aborted after the learning loop in Algorithm 1 seemed to not have been iterated any more. Since the testing is done in parallel to the learning, this is not ultimately true because the generated hypotheses were used for test case generation and none of these yielded an error or warning. Although, the confidence about the accuracy of the system is not as high as for a correctly learned hypothesis model.

Membership queries

Internally, the BBW program processes input data with a delay and uses feedback input data from earlier time slices to compute the output data. As a Mealy automaton cannot specifically cope with a time dimension, this has to be compensated by adding further states. Thus, the Mealy automaton model grows dramatically during the attempt to fully learn the SUT as the Table 4.7, Table 4.8 and Table 4.9 show with the hypothesis size. Figure 4.22 show that the number of membership queries, resulting from the learning loop depicted in Algorithm 1, grows polynomially with the number of iterations. For example, this can be explained with the consistency property from Formula 3.3 and the implementation thereof 3.1.3.(2). For a single new suffix $s \in S$, the learner has to request $|BLUE|$ values using membership queries.

Equivalence queries

It becomes evident that complete learning does not make much use of the learning environment in LBTest. Looking at the data in Table 4.7, the L* Mealy learner rarely makes use of equivalence queries. The properties that guide the learner to internally resolve which input sequences are important (described in Section 2.1) are the reasons for this behaviour. For instance, Table 4.7 shows that the model checker cannot find any contradictory test sequences leaving it to the random test generator to find a counterexample to the learner's hypothesis. These counterexamples are not guaranteed to be minimal (since they are randomly chosen) as in the previous section thus, extending the learning time following Formula 2.1. As the learner only consults the model checker very rarely, it is possible that the learner explores a known equivalence class instead of discovering new ones via the valuable information from the counterexamples that can be integrated much faster than lengthy internal investigations. The charts in Figure 4.23 indicate that the L* Mealy learner stays with increasing frequency in the learning loop instead of performing an equivalence query. This effect becomes most evident for requirement 3, seen in Figure 4.23c, which also has the steepest slope in the number of membership queries.

Membership queries per equivalence query

The Figure 4.24 implies a polynomial increase in the number of membership queries per equivalence query which is most distinct in the graph in Figure 4.24c. This behaviour can easily be deduced from the previous discussion of the individual trends of the number of membership queries and the number of equivalence queries. It demonstrates the issue that the learner only builds few hypotheses.

5.3. Possible optimizations

The implementation of both L* and L* Mealy can be optimized to further reduce the number of queries and the total learning time. According to profiled learning runs, it was most time-consuming to check the observation table against the consistency and completeness property. Instead of globally validating Formula 3.2 and Formula 3.3, only those local prefixes could be checked for which the conditions (for example a new observation was inserted) have changed.

The data in the observation table of the implemented L* learner occupies up to $O(|Q|^4)$ bits in the memory according to Formula 3.1 and Angluin's conclusions in [Ang87]. The L* Mealy learner needs even more space because of the output vectors compared with the binary output used in L*. Thus, especially for big automata, the operations on the observation table heavily influence the run time of the learners. Therefore, the learning process can be accelerated with optimized data structures and fewer table lookups — e.g. making use of caching, hashing of input sequences or referencing values instead of copying them between tables.

The learning time could be reduced if it was possible to steer the setup and tear down process of the SUT. In that case, sequences with equal prefixes would not require a full restart of the SUT. This can for instance be achieved with a more complex wrapper for the SUT that forwards more information from the execution. An approach to reducing the number of membership queries might be to enhance the quality of the counterexamples from the teacher so that the learner gets as much information on the model as possible. It is conceivable to introduce a measure of sufficient equivalence between the hypotheses and the model that should be learned as proposed by Berg et al. [Ber+03]. This suggestion relaxes the strictness of a complete learner and might save many queries and execution time.

6. Conclusion

In this work, a working implementation of the two automata learning algorithms L* and L* Mealy was accomplished. A simple example in Section 4.1.1 demonstrated that learning a model without knowledge of the actual representation is possible. Empirical data from the L* evaluation framework was helpful to understand the behaviour of the learner for varying alphabet and state space sizes. This information was also essential to grasp the structure of the black-box software in the case studies. The integration of L* Mealy into the learning-based testing environment LBTest helped to assess the feasibility of a complete learner in the area of learning-based testing.

L*

The conducted empirical study, as presented in Section 4.1 and discussed in Section 5.1, indicates a much better practical performance than the theoretically implied run time in Formula 2.1. The main findings drawn from the evaluation framework for Angluin's L* algorithm are listed below:

1. The more the accepting states ration, as defined in Section 4.1, deviates from 50% the more queries are needed to fully learn the DFA.

2. The essential measured parameter, that determines the learning time, is the number of membership queries.

3. The number of equivalence queries as a function of the state space size is bounded by a linear function and, in case of easy-to-learn automata, grows logarithmically.

4. Counterintuively, the alphabet size of the randomly generated DFAs does not contribute to the growth in the number of membership queries.

5. The L* algorithm converges nearly linearly in the state space size against a correct hypothesis in case of an easy-to-learn automaton.

6. Even for hard-to-learn DFAs the number of membership queries seems to be bound by $O(|\Sigma||Q|^2)$ in the average case rather than $O(|\Sigma||Q|^3)$.

7. The number of membership queries per equivalence query grows linearly as a function of the alphabet and the state space size.

L* Mealy

With the integration of the L* Mealy algorithm into LBTest and its evaluation using the *Cruise Controller* application, it was demonstrated that complete learning algorithms can be successfully applied to learning-based testing.

The case study *Break-By-Wire* software clearly demonstrated that the algorithm does not scale for larger applications. The results showed a polynomial growth in the number of membership queries during the learning process which lead to very slow learning as they have to be executed on an external system with potentially exhaustive executions. As discussed for L*, the growing gap between the number of equivalence queries and the number of membership queries leads to time-consuming calculations in LBT. Hence, a clear drawback of a complete learner is the inability to make extensive use of the model checker using the hypotheses from equivalence queries like the inbuilt IKL incremental learner that repeatedly checks back on it. Overall, the L* Mealy algorithm is not well-suited for learning-based testing.

Limitations

More industrially used software components would have made it possible to generalize conventional behaviour of the L* Mealy algorithm in LBT. If there had been more time for this research, the statements could have been substantiated with empirical data from extreme range values for the state and the alphabet size. Additionally, an evaluation framework for the L* Mealy algorithm similar to the present implementation of L* would have been implemented to confirm the observations in LBTest.

Perspectives

Further research needs to be devoted to both minimizing the number of membership queries in order to more quickly differentiate between the required equivalence classes of input vector sequences and incorporating information from the model checker that has access to descriptive model requirements. With this combination, learning-based testing would become applicable to bigger software applications. Additionally, the structure of the modelled systems has to be studied with a specific focus on the difficulty for a learner to distinguish between equivalence classes to apply further optimizations.

Bibliography

[Ang87] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples.* Tech. rep. Department of Computer Science, Yale University, 1987, pp. 87–106.

[Ber+03] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayak Saksena. *Insights to Angluin's Learning.* Tech. rep. Department of Computer Systems, Uppsala University, 2003.

[Che13] Yingke Chen. "Machine Learning-based Verification: Informed Learning of Probabilistic System Models". PhD thesis. Department of Computer Science, Aalborg University, 2013.

[DR08] Amit Deshpande and Dirk Riehle. "The Total Growth of Open Source". In: Open Source Systems. Springer, 2008, pp. 197–209.

[Fen+13] Lei Feng, Karl Meinke, Fei Niu, Mudassar A. Sindhu, and Peter Y. H. Wong. *Case Studies in Learning-based Testing.* Tech. rep. School of Computer Science and Communication, Royal Institute of Technology, 2013.

[GA07] Franz Wotawa Gordon Fraser and Paul E. Ammann. *Testing with model checkers: a survey.* Tech. rep. Graz University of Technology, 2007.

[Gol78] E Mark Gold. "Complexity of automaton identification from given data". In: *Information and Control* 37.3 (1978), pp. 302–320.

[GPY06] Alex Groce, Doron Peled, and Mihalis Yannakakis. "Adaptive Model Checking". In: *Logic Journal of IGPL* 14.5 (2006), pp. 729–744.

[How12] Falk M. Howar. "Active learning of interface programs". PhD thesis. Department of Computer Science, Technical University Dortmund, 2012.

[Mea55] George Mealy. *A Method for Synthesizing Sequential Circuits.* Tech. rep. Massachusetts Institute of Technology, 1955.

[MNS12] Karl Meinke, F. Niu, and M. Sindhu. "Learning-Based Software Testing: A Tutorial". In: *Leveraging Applications of Formal Methods, Verification, and Validation.* Ed. by Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, pp. 200–219.

[MS11] Karl Meinke and Muddassar A. Sindhu. "Incremental learning-based testing for reactive systems". In: *Tests and Proofs.* Ed. by M. Gogolla and B. Wolff. Vol. 6706. Lecture Notes in Computer Science. Springer, 2011, pp. 134–151.

[MS13] Karl Meinke and Muddassar A. Sindhu. *LBTest: A Learning-based Testing Tool for Reactive Systems.* Tech. rep. School of Computer Science and Communication, Royal Institute of Technology, 2013.

[Nie03] Oliver Niese. "An Integrated Approach to Testing Complex Systems". PhD thesis. Department of Computer Science, Dortmund University, 2003.

[Nor09] Daphne A. Norton. “Algorithms for Testing Equivalence of Finite Automate, with a Grading Tool for JFLAP”. MA thesis. Department of Computer Science, Rochester Institute of Technology, 2009.

[NP10] John Derrick Neil Walkinshaw Kirill Bogdanov and Javier Paris. “Increasing Functional Coverage by Inductive Testing: A Case Study”. In: *Testing Software and Systems*. Ed. by José Carlos Maldonado Alexandre Petrenko Adenilso Simão. Vol. 6435. Lecture Notes in Computer Science. Springer, 2010, pp. 126–141.

[NT74] K.S. Narendra and M. Thathachar. “Learning Automata - A Survey”. In: *Systems, Man and Cybernetics, IEEE Transactions on* SMC-4.4 (July 1974), pp. 323–334.

[RS09] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. 3rd ed. Springer, 2009.

[SB04] Klaus Schneider and Wilfried Brauer. *Verification of Reactive Systems : Formal Methods and Algorithms*. Springer, 2004.

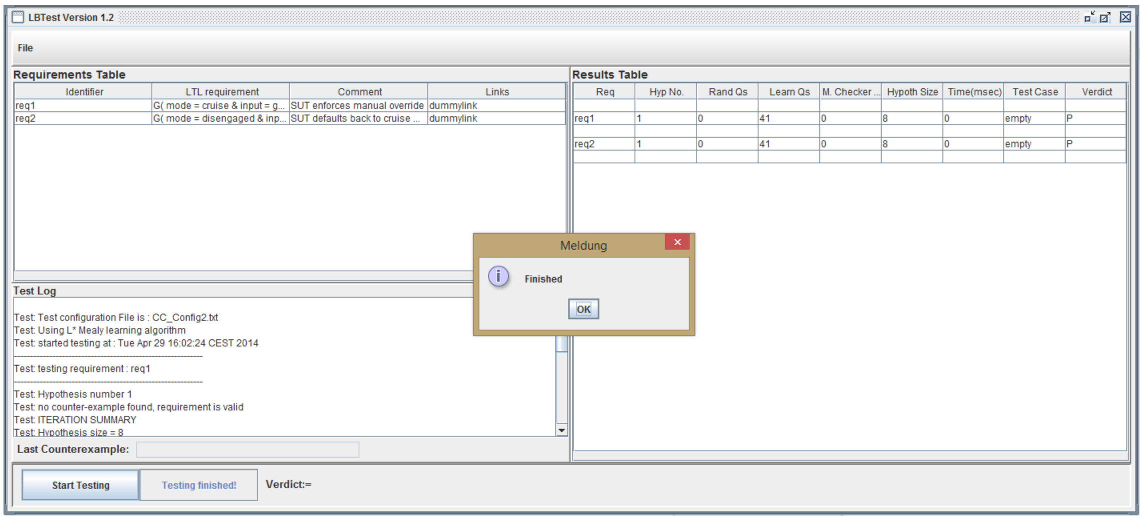
[Sin13] Muddassar Azam Sindhu. “Algorithms and Tools for Testing of Reactive Systems”. PhD thesis. Department of Computer Science and Communication, Royal Institute of Technology, 2013.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. 2nd ed. Thomson, 2006.

[SM08] Peter Sanders and Kurt Mehlhorn. *Algorithms and Data Structures : The Basic Toolbox*. Springer, 2008.

Appendix

A. LBTest



Requirements Table

Identifier	LTL requirement	Comment	Links
req1	G(mode = cruise & input = g...	SUT enforces manual override	dummylink
req2	G(mode = disengaged & inp...	SUT defaults back to cruise ...	dummylink

Results Table

Req	Hyp No.	Rand Qs	Learn Qs	M. Checker ...	Hypoth Size	Time(msec)	Test Case	Verdict
req1	1	0	41	0	8	0	empty	P
req2	1	0	41	0	8	0	empty	P

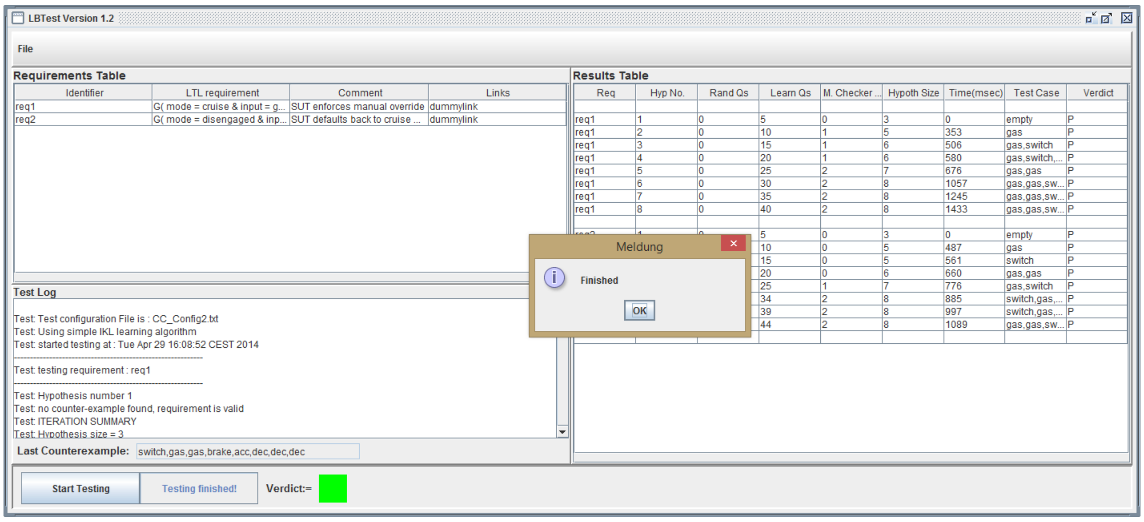
Test Log

```
Test: Test configuration File is : CC_Config2.txt
Test: Using L* Mealy learning algorithm
Test: started testing at : Tue Apr 29 16:02:24 CEST 2014
------------------------------------------------------------
Test: testing requirement : req1
------------------------------------------------------------
Test: Hypothesis number 1
Test: no counter-example found, requirement is valid
Test: ITERATION SUMMARY
Test: Hypothesis size = 8
```

Last Counterexample:

Verdict:=

Figure A.1.: A screenshot taken after LBTest finished the learning process in the case study *Cruise Controller* with the L* Mealy learner



Requirements Table

Identifier	LTL requirement	Comment	Links
req1	G(mode = cruise & input = g...	SUT enforces manual override	dummylink
req2	G(mode = disengaged & inp...	SUT defaults back to cruise ...	dummylink

Results Table

Req	Hyp No.	Rand Qs	Learn Qs	M. Checker ...	Hypoth Size	Time(msec)	Test Case	Verdict
req1	1	0	5	0	3	0	empty	P
req1	2	0	10	1	5	353	gas	P
req1	3	0	15	1	6	506	gas,switch	P
req1	4	0	20	1	6	580	gas,switch,...	P
req1	5	0	25	2	7	676	gas,gas	P
req1	6	0	30	2	8	1057	gas,gas,sw...	P
req1	7	0	35	2	8	1245	gas,gas,sw...	P
req1	8	0	40	2	8	1433	gas,gas,sw...	P
req2	1	0	5	0	3	0	empty	P
[illegible]	[illegible]	[illegible]	10	0	5	487	gas	P
[illegible]	[illegible]	[illegible]	15	0	5	561	switch	P
[illegible]	[illegible]	[illegible]	20	0	6	660	gas,gas	P
[illegible]	[illegible]	[illegible]	25	1	7	776	gas,switch	P
[illegible]	[illegible]	[illegible]	34	2	8	885	switch,gas,...	P
[illegible]	[illegible]	[illegible]	39	2	8	997	switch,gas,...	P
[illegible]	[illegible]	[illegible]	44	2	8	1089	gas,gas,sw...	P

Test Log

```
Test: Test configuration File is : CC_Config2.txt
Test: Using simple IKL learning algorithm
Test: started testing at : Tue Apr 29 16:08:52 CEST 2014
------------------------------------------------------------
Test: testing requirement : req1
------------------------------------------------------------
Test: Hypothesis number 1
Test: no counter-example found, requirement is valid
Test: ITERATION SUMMARY
Test: Hypothesis size = 3
```

Last Counterexample: switch,gas,gas,brake,acc,dec,dec,dec

Verdict:= 🟩

Figure A.2.: A screenshot taken after LBTest finished the learning process in the case study *Cruise Controller* with the inbuilt IKL learner

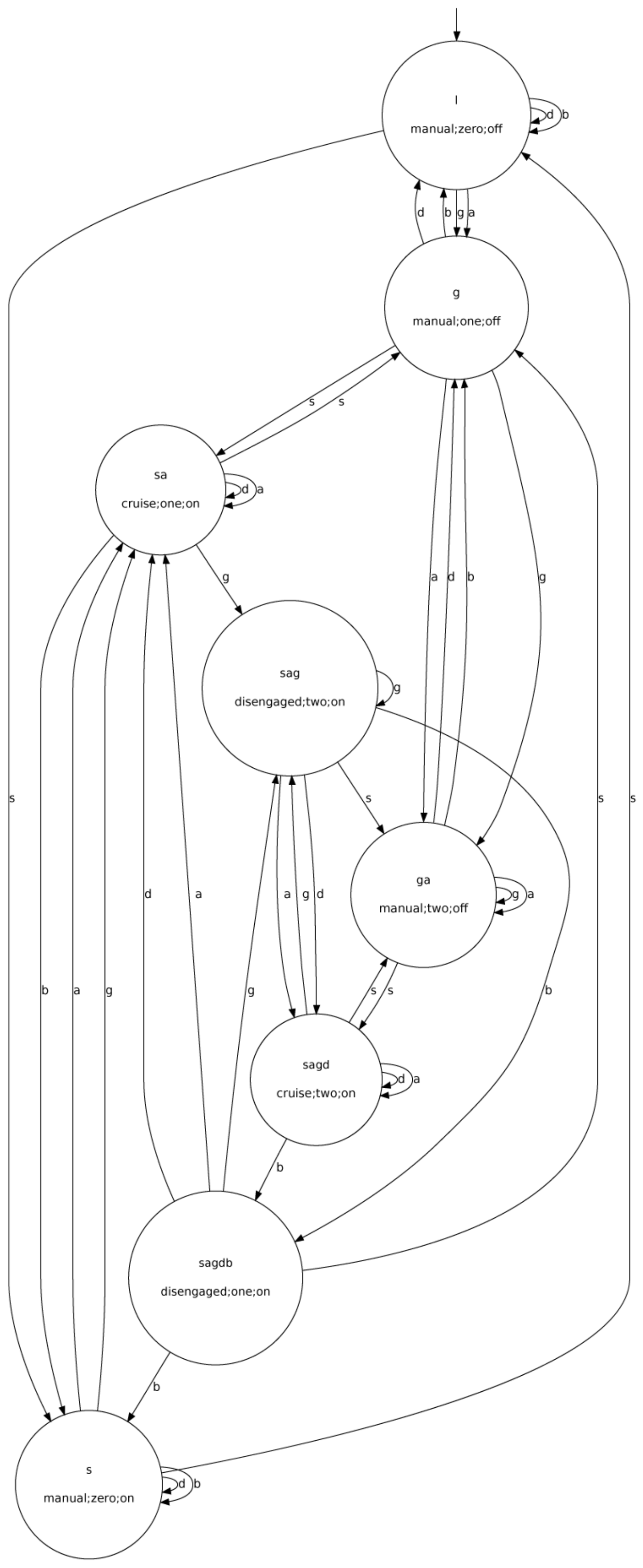
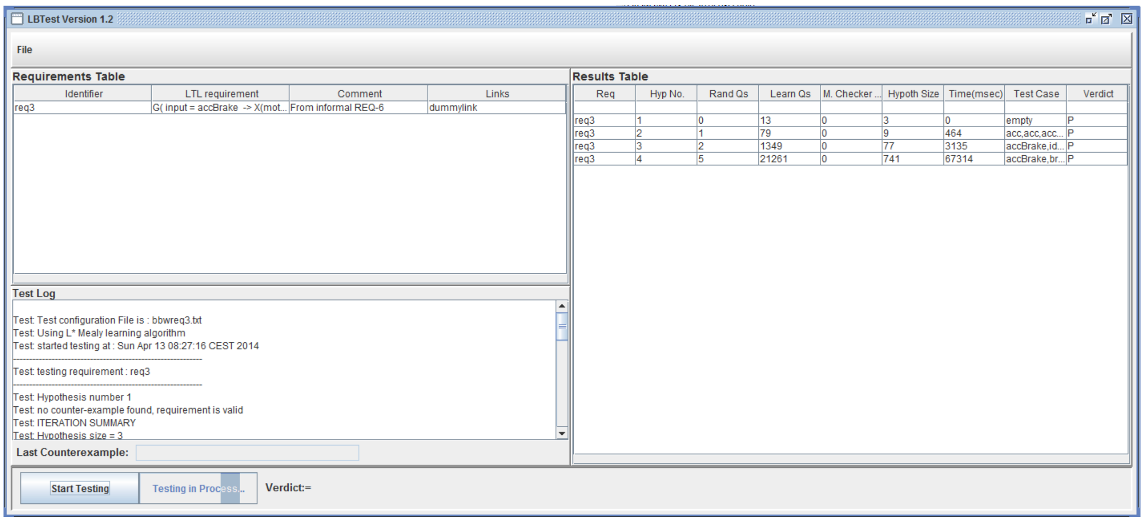


Figure A.3.: The Mealy automaton of the successfully learned *Cruise Controller* software with LBTest using L* Mealy



Requirements Table

Identifier	LTL requirement	Comment	Links
req3	G(input = accBrake -> X(mot...	From informal REQ-6	dummylink

Test Log

```
Test: Test configuration File is : bbwreq3.txt
Test: Using L* Mealy learning algorithm
Test: started testing at : Sun Apr 13 08:27:16 CEST 2014
------------------------------------------------------------
Test: testing requirement : req3
------------------------------------------------------------
Test: Hypothesis number 1
Test: no counter-example found, requirement is valid
Test: ITERATION SUMMARY
Test: Hypothesis size = 3
```

Last Counterexample:

Results Table

Req	Hyp No.	Rand Qs	Learn Qs	M. Checker ...	Hypoth Size	Time(msec)	Test Case	Verdict
req3	1	0	13	0	3	0	empty	P
req3	2	1	79	0	9	464	acc,acc,acc...	P
req3	3	2	1349	0	77	3135	accBrake,id...	P
req3	4	5	21261	0	741	67314	accBrake,br...	P

Figure A.4.: A screenshot taken while the LBTest tool tests the *Break-By-Wire* software against Requirement 3 with the L* Mealy learner