Thesis

Exploiting structure in counterexamples to speed up equivalence checking in the minimally adequate teacher framework

Tom Catshoek

Active Learning

TU Delft

Thesis

Exploiting structure in counterexamples to speed up equivalence checking in the minimally adequate teacher framework

by

Tom Catshoek

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday January 21, 2021 at 14:00.

Student number:	4311914	
Project duration:	October 2019 – January 2021	
Thesis committee:	Prof. dr. ir. R. L. Lagendijk,	TU Delft, Chair of Committee
	Dr. ir. S. E. Verwer,	TU Delft, Supervisor
	Dr. ir. M. Aniche,	TU Delft, Committee Member

TU Delft Delft University of Technology

Abstract

Active state machine learning algorithms are a class of algorithms that allow us to infer state machines representing certain systems. These algorithms interact with a system and build a hypothesis of what the state machine describing that system looks like according to the behavior they observed. Once the algorithm arrives at a hypothesis, it sends it to a so-called equivalence oracle to be checked. The equivalence oracle returns a counterexample trace describing different behavior between the hypothesis and the real system if one exists. Traditionally, this task is done using a technique called the W-method, which provides certain guarantees about finding a counterexample if one exists, but this is rather slow due to the thorough search it needs to do. In real systems, there are sometimes patterns to the counterexamples to be found during the learning process, which can be exploited to speed up the process of finding new counterexamples. Instead of performing an exhaustive search each time, we first use the already observed patterns to generate new traces to try before falling back to traditional techniques in case no counterexample was found in the first step. This allows us to find counterexamples more quickly than usual, and because we can always fall back on the techniques that do provide guarantees we do not have to sacrifice correctness. In the same amount of time, our method provides an up to 26x increase in the number of states found compared to the W-method, while remaining fully black box and without degrading worst-case performance. We also show that, when not working in a fully black-box scenario, fuzzing can be a successful complementary technique to apply next to active learning. However, we also show that relying on fuzzing alone does not always fully capture the behavior of a system, and when correctness is critical such as in the case of model checking, it needs to be supplemented with additional techniques to result in fully correct models. Finally, we use our home-grown state machine learning library to participate in the 2020 RERS challenge, and apply the techniques we studied so far to attain a perfect score on the sequential LTL track.

Tom Catshoek
Delft, January 2021

Preface

Have you ever heard the saying "May you live in interesting times"? Well, 2020 was an *interesting* year to say the least. Due to the coronavirus pandemic, the university closed and we all had to adapt to working from home. Not only did working together become more difficult, but we were also all unable to physically go and see friends. Now I am naturally not the biggest social butterfly, but I would be lying if I said this did not affect me at least a little bit. At times it was difficult to stay focused and motivated, but everyone in our master group helped each other persevere even though we could only see each other online. Next to the weekly video meetings where everyone kept each other up to date on their work, the game nights we organized every other week were a very welcome distraction from the isolation we undoubtedly all experienced.

First of all, I would like to thank my supervisor, Sicco Verwer for his invaluable guidance and feedback and for convincing me to keep going when I felt like my work was going nowhere at times. Had we not had our meetings and the ability to bounce ideas off each other I would most likely not have gotten this far in my work. I would also like to thank Chris Hammerschmidt and Clinton Cao for always being up for a chat, whether work-related or not, and the fun matches of Quake 3 we played. Furthermore, I would like to thank Daniël Vos for getting a TU Delft CTF team off the ground and organizing the meetings in which we all learned valuable skills from each other and had a lot of fun. I also thank the rest of our master group for always being there to talk to and exchange feedback. And last but not least, I would like to thank my parents for their support throughout all my years at TU Delft, and for temporarily welcoming me back home when sitting alone in my studio on campus during lockdown started wearing me down.

Tom Catshoek
Delft, January 2021

Contents

1 Introduction

To understand the world around us, it is important to have a mental model of how it functions. For example, kids seem to have a talent for figuring out how electronic devices work by pushing buttons and observing what happens. We can do the same in software: by interacting with a system, observations can be made, and a model consistent with those observations can be inferred. These models can be represented as state machines, with the states corresponding to the states of the system, and transitions between those states representing the actions that can be taken from them. Model inference techniques that actively interact with a system to infer how it works are *active state machine learning* algorithms.

However, it is not a strict requirement to actively interact with a system to learn how it works. If a set of passively observed interactions or traces is available, those can also be used to infer a state machine consistent with those observations. These techniques that rely on a predefined set of traces are *passive state machine learning* algorithms. However, as in this case there is no active interaction with the system that is being modeled, the resulting model will only be as accurate as the given data allows. If a certain behavior of the system does not appear in the set of observed interactions, this behavior will also be absent from the learned model.

As one can imagine, being able to automatically construct a model that describes the behavior of a given system can be useful in many situations. For example, in software testing, it can be useful to check if a given system conforms to a formal specification. For a real world example of this we can look at a pilot study done at ASML, where they investigate the feasibility of using state machine learning on the software running their lithography machines [41]. Other notable examples include the automated reverse engineering of a botnet C&C protocol [12], and more general black box testing such as checking implementations of TLS [14], SSH [16], and OpenVPN [13] protocols. Previous work has also used state machine learning techniques to detect security flaws in mobile phone apps [40], banking cards [1] and their readers [8], as well as industrial control systems [27].

A downside of active state machine learning techniques is that, because they interact with a running system and pose a large amount of queries to get a complete overview of the behavior of it, they can take a long time to run. Previous work has shown that when the total time of the learning process increases, the time spent testing hypotheses (equivalence queries) dominates the total time spent learning [41]. Therefore, this seems like a prime target to focus on if we want to speed up the process of active state machine learning.

Active learning

Regular inference, or the process of learning a regular language from sets of negative and positive examples is a well studied problem in computer science. The first algorithm to be able to complete this task in polynomial time (in the size of the system under learning) was published in 1987 by Dana Angluin [3]. This algorithm, named L*, is presented in a framework called "minimally adequate teacher", or MAT. A schematic overview of the MAT framework can be seen in Figure 1.1.

The MAT framework consists of a learner component and a teacher component. The learner poses two types of queries to the teacher, membership and equivalence queries. For membership queries, the learner poses the teacher a question in the form of an input sequence. It is assumed the teacher has access to the actual system under learning (SUL) in the form of a DFA, so it can easily run the given input sequence and see the corresponding output. The teacher answers the query with either true or false, depending on if the input is in the regular set representing SUL or not. After having posed enough membership queries, the learner is able to construct a hypothesis representing what it thinks the model looks like given the information it has so far. Then, it asks the teacher if this hypothesis is correct in the form of an equivalence query. This is where the Teacher's equivalence oracle comes in. This oracle is able to determine if there exists a counterexample proving the inequivalence of the learner's hypothesis to the actual SUL. If the hypothesis is correct, the teacher says so and the learning stops. If the hypothesis is incorrect, the teacher provides the learner a counterexample that highlights a case where the hypothesis and the actual system give a different output. The learner then adds the information of this counterexample to its observations and continues asking membership queries to construct a new hypothesis.

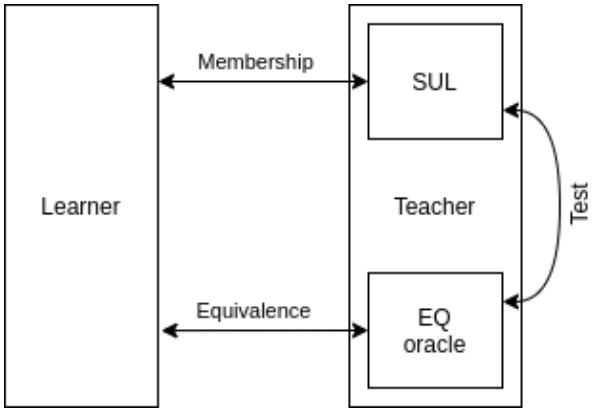

Figure 1.1: Schematic overview of the minimally adequate teacher framework

Of course, in reality we do not have access to an equivalence oracle that magically gives us a counterexample if one exists, so we need to resort to conformance testing techniques to approximate it. These techniques can be used to generate a set of test inputs to run on both the hypothesized model and the actual SUL, and then if both systems give the same output to all test sequences we can conclude that they are equal. Several methods of generating these sets of test sequences exist, with the most notable ones being the W-method [10] and Wp-method [17] which use the access sequences to all states, concatenated with a repeated product of the input alphabet which in turn is appended by the characterizing set of the hypothesized model. For a more formal and complete explanation please refer to chapter 2. Others include the Transition Tour method, the Distinguishing Sequence method, and the Unique Input-Output Sequence method. [39]

Since the introduction of L* and the MAT framework in 1987, several improvements over the classic L* algorithm have been proposed. A nice property of the MAT framework is that the individual components can be swapped out as long as they can fulfill the same task, perhaps in a more efficient way. TTT [6], which to the best of our knowledge is the current state of the art active state machine learning algorithm, swaps out the learner component of the classic L* for a more efficient implementation which uses discrimination trees to avoid asking redundant membership queries. TTT also makes clever use of a trie datastructure to efficiently store discriminating sequences, further improving efficiency.

RERS

One of the driving forces behind the work in this thesis is our participation in the RERS (rigorous examination of reactive systems) challenge. The RERS challenge is a competition that invites its participants to solve various problems in two classes: reachability and LTL (linear temporal logic). The reachability track consists of problems concerning obfuscated source code of reactive systems, with the goal being to find out which error states in the program are reachable and which are not. The given source code compiles into a program that can be communicated with over stdin/stdout, where each input can be seen as an action that puts the system in a different state and causes it to give an output. It seems like a natural fit to model this using Mealy machines (Def. 2.1.4), and so that is the approach taken in this work. However, we will see that as soon as the programs become anything other than trivial, a naive approach quickly runs into problems for various reasons. We will attempt to identify these problems and figure out how to work around them to make active state machine learning a feasible strategy for as many RERS challenges as possible.

1.1. Outline

In this thesis, we describe how to infer models of software using active state machine learning, and investigate the performance characteristics of the most commonly used algorithms. We investigate methods to reduce their run time in practice, thus opening up more problems to feasible application of state machine learning techniques. We test our approach in a real-world setting by participating in the RERS challenge and describe our method of solving both the reachability and LTL problems. In short, we aim to answer the following three research questions:

- **How can we speed up the process of finding counterexamples during equivalence checking?** A vital part of state machine learning algorithms based on the minimally adequate teacher framework is the equivalence oracle. In practice, such an oracle can be realized using various algorithms which all have certain tradeoffs. The theory of equivalence checking is a well researched area, so a summary of the current work is provided in section 2.4. First, we show that it is very much worthwhile to spend some time engineering a proper method of communicating with the system you are trying to learn from, and propose an improved version of the W-method equivalence checker in section 3.3. After this, we propose two new equivalence checking methods which aim to speed up the process of finding counterexamples by exploiting their structure in chapter 4.
- **How can we use white/grey box testing to improve the effectiveness of state machine learning algorithms?** To the best of our knowledge, all current active state machine learning algorithms are black box in the sense that they only operate on inputs and outputs of a given system and do not utilize any source code analysis or instrumentation. Previous work has shown that combining learning and fuzzing [24] can be very effective in certain cases. In chapter 5 we investigate this method further, show that by itself it is possibly insufficient to capture the complete behavior of a system, and apply it to the RERS challenge problems.
- **What is the most effective set of techniques out of the ones we consider to solve the RERS challenges in the sequential reachability and LTL tracks?** After investigating the above methods of speeding up active state machine learning, in chapter 6 we investigate what the most effective techniques are for both the reachability and the LTL track of the RERS challenge 2020, and evaluate our approach on the problems of RERS 2019.

1.2. Main Contributions

- **Python state machine learning library**. One of the main contributions of this thesis is a new, easy to use active state machine learning library written in python. It aims to fulfill the same goals as the well-known LearnLibMealyisberner2015open[1] library, but focuses on ease of use, simplicity, and easy extensibility. Of course, it is not as battle-tested as LearnLib and does not support all the same algorithms yet, but the ease of experimenting with new code and ideas python gives us makes it a worthwhile tradeoff for academic purposes, or at least this thesis. Most of the algorithms mentioned in this thesis have been implemented by the author, and have been used over the course of this thesis. They have also been used to solve the problems of the 2020 RERS challenge.

  At the time of writing, the code used in this thesis can be found here [2]. A neatly packaged library will follow at a later date, with a link to it available on this page.

- **Mutating equivalence checker**. Standard equivalence checking techniques like the W-method are thorough, but slow. We propose a new equivalence checking step that exploits structure in the counterexamples found so far to perform a more targeted search for new ones. This technique provides a significant speedup on the RERS problems we tested it on, and can be stacked with the W-method to still provide the same correctness guarantees.

- **Fuzzing**. We confirm the previous work which showed us fuzzing can be combined with learning to greatly improve the rate at which new states are discovered. However, we also show that only using the traces found by fuzzing can be insufficient to capture the complete behavior of a system. We again propose to stack the fuzzer-based equivalence check with other methods to make sure nothing is missed.

- **RERS**. We devise a strategy to solve both the sequential reachability and LTL tracks of RERS 2020. We integrate the NuSMV model checker into our library and succesfully use it to perform LTL checks on the RERS problems. We also show that while active state machine learning can be a powerful technique in black box scenarios, it has trouble competing with white/greybox fuzzing for pure reachability tasks. However, for LTL tasks, learning state machines is a feasible approach and our method manages to attain a perfect score on both the RERS 2019 and 2020 sequential LTL tracks.

[1] https://learnlib.de/
[2] https://github.com/TCatshoek/lstar

2

Preliminaries and Related Work

In this chapter we will introduce the reader to some knowledge required to understand what active state machine learning algorithms do, and how they function. We will explain the two types of models we use, DFAs and Mealy machines. DFAs are the model of choice in the original L* algorithm as well as TTT. Both algorithms have been adapted to learn the more expressive Mealy machine as well, which we use to model the RERS problems. We will introduce the minimally adequate teacher framework, in which most active learning algorithms are defined, and dive into the inner workings of L* and TTT. After that we will explain equivalence oracles, which are a necessary component in the MAT framework and attempt to prove or disprove equivalence of the proposed hypothesis model to the system under learning (SUL). Lastly, we will look at some related work which we draw inspiration from.

2.1. Automata

The hypotheses created by state machine learning algorithms usually come in the form of automata, such as DFAs or Mealy machines. More complex and expressive automata like for example register [23] or visibly pushdown automata [2] are also possible to learn, but to narrow down the scope of this thesis we will focus on DFAs and Mealy machines. We focus on DFAs since both the L* and TTT algorithms were originally developed with the goal of learning regular languages (and thus DFAs) in mind, and Mealy machines since the RERS reachability problems are well suited to be represented by them. We aim to stick closely to the notation used in current literature.

2.1.1. DFAs

Definition 2.1.1. (DFA) A deterministic finite automaton U is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where:

- Q is the set of all states
- Σ is the input alphabet
- q_0 is the initial state
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- F is the set of accepting states

Furthermore, the language accepted by U is defined by $L(U) = \{w \in \Sigma^* | \delta(q_0, w) \in F\}$

To clarify, Figure 2.1 shows an example DFA $\mathcal{A}$ which accepts the regular language defined by the regular expression b*a+b[ab]*. The three circles marked q0, q1 and q2 represent the states Q, and the input alphabet $\Sigma = \{a, b\}$. The labeled arrows represent the transitions in δ, and the double circle around q_2 marks it as an accepting state.

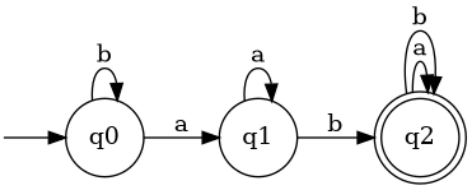

Figure 2.1: An example DFA

To get the state q' after a certain transition from another state q we can write $q' = \delta(q,a)$ with $a \in \Sigma$ and $q, q' \in Q$. The state q' in this case is called the a-successor of state q. We can slightly abuse this notation to extend the transition function to $q' = \delta(q,w)$ so it can accept words $w \in \Sigma^*$, where Σ^* is the concatenation of any number of elements of Σ of any length. If we define the concatenation of a symbol a to a word w as $a \cdot w$ (or simply aw), and ϵ as the empty word, δ can be defined inductively as

Definition 2.1.2. (DFA transition function)

$$\begin{aligned} \delta(q,\epsilon) &= q && \forall q \in Q \\ \delta(q,aw) &= \delta(\delta(q,a),w) && \forall q \in Q, a \in \Sigma, w \in \Sigma^* \end{aligned}$$

An example in the case of the DFA $\mathcal{A}$ could be $q_2 = \delta(q_0, ab)$, where ab is the input that takes us from state q_0 to the accepting state q_2.

In the case of DFAs, it is also convenient to define a state output function $\lambda : \Sigma^* \to \mathbb{B}$:

Definition 2.1.3. (DFA state output function)

$$\lambda(w) = \begin{cases} 1 & \text{if } \delta(q_0,w) \in F \\ 0 & \text{otherwise} \end{cases} \quad \forall w \in \Sigma^*$$

The full formal description of this DFA $\mathcal{A} = (Q, q_0, F, \delta, \Sigma)$ is

- All states $Q = \{q_0, q_1, q_2\}$
- q_0 is the starting state
- The accepting states $F = \{q_2\}$
- The transition function δ can be represented by the transition table:

	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

- The input alphabet $\Sigma = \{a, b\}$

2.1.2. Mealy Machines

A Mealy machine is much like a DFA in the sense that it has states and transitions which bring it to a new state through input symbols. However, it does away with the notion of accepting states and instead defines an output function, which for each transition in the graph defines an output symbol.

Definition 2.1.4. (Mealy machine) a Mealy machine is a 6-tuple $(Q, \Sigma, \Omega, q_0, \delta, \gamma)$, where:

- Q is the set of states
- Σ is the input alphabet
- Ω is the output alphabet
- q_0 is the initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $\lambda : Q \times \Sigma \to \Omega$ is the output function

Figure 2.2 shows an example Mealy machine $\mathcal{M}$ with the same general structure as the DFA in Figure 2.1. However, note that no state is marked as being an accepting state, and each transition now not only has an input symbol, but also an output symbol.

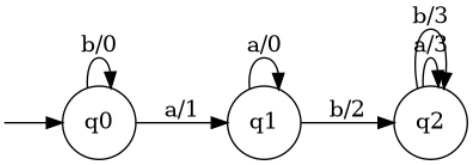

Figure 2.2: An example Mealy machine

The transition function for Mealy machines is defined analogously to the one for DFAs (2.1.2), and the output function is similar as well. The output function λ can be defined inductively as

Definition 2.1.5. (Mealy machine output function)

$$\begin{aligned} \lambda(q, \epsilon) &= \epsilon && \forall q \in Q \\ \lambda(q, a \cdot w) &= \lambda(q, a) \cdot \lambda(\delta(q, a), w) && \forall q \in Q, a \in \Sigma, w \in \Sigma^* \end{aligned}$$

An example input for Mealy machine $\mathcal{M}$ could be ab, which would take us to state q_2 from state q_0 and through state q_1, while giving us the output word 12.

The full formal definition of Mealy machine $\mathcal{M} = (Q, \Sigma, \Omega, q_0, \delta, \lambda)$ is

- The set of all states $Q = \{q_0, q_1, q_2\}$
- The input alphabet $\Sigma = \{a, b, c\}$
- The output alphabet $\Omega = \{1, 2, 3\}$
- The initial state q_0
- The transition function δ represented by the transition table

	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_2	q_2

- The output function λ represented by the table

	a	b
q_0	1	0
q_1	0	2
q_2	3	3

2.2. Minimally Adequate Teacher framework

As discussed in the introduction, most active state machine learning algorithms are based on the minimally adequate teacher framework proposed by Angluin [3] in 1987. In this paper, Angluin describes the minimally adequate teacher as a component in the framework that can answer two types of questions from the learner: membership queries containing a string t, and conjectures (or equivalence queries) consisting of a description of a regular set S. Since the original L* algorithm is intended to learn regular sets (which can be represented by DFAs), the answer to a membership query is simply yes or no, depending on if t is in the unknown set or not. The answer to an equivalence query is either yes if the conjectured set is correct, or a counterexample from the symmetric difference of S and the unknown language if not.

2.3. Learners

The learner component in the MAT framework is the part that asks membership queries to the teacher, and when it has collected enough information, constructs a hypothesis which it then poses to the teacher in the form of an equivalence query. If no counterexample is found, learning stops. If a counterexample is found, it is handed back to the learner, which uses the information gained from the counterexample to refine its hypothesis, and the cycle begins again until no counterexample is found anymore.

The learner stores the result of all observations it made in an internal data structure, and calculates what new membership queries to pose based on what knowledge it already has. The specifics of this process depend on the type of learning algorithm chosen, which will be discussed in the following sections.

A generic high level description of the "learning loop" looks as follows:

input : A leaner L and a teacher T
output: A hypothesis H matching the unknown system

$C \longleftarrow \epsilon$ `// Keep track of counterexample C`

while $C \neq None$ **do**
  $C \longleftarrow$ None
  $H \longleftarrow refineHypothesis(L)$ `// Asks membership queries until L can make a new H`
  $C \longleftarrow equivalenceQuery(T, H)$ `// Returns a counterexample if there exists one`
  if $C \neq None$ **then**
    $processCounterexample(L, C)$ `// Lets the learner process the counterexample`
  else
    return H
  end
end

Algorithm 1: High level learning loop

Assuming a perfect equivalence oracle (one that always gives a counterexample if one exists within reasonable time; we will later see that this is a problem in practice), and a competent learner, it is easy to see that the hypothesis eventually returned will match the unknown system since there exists no counterexample in which it's behaviour differs from the unknown system.

To make this idea workable in practice, we need implementations of both the learner and the equivalence oracle, which we will discuss in the following sections.

2.3.1. L*

The L* algorithm [3] is the first published algorithm to be able to learn an unknown regular set from a teacher in polynomial time. It does this by making use of a data structure called an **observation table**. For the following section, let U be an unknown regular set, and A a set describing the finite alphabet containing all the characters needed to create every word in U.

Definition 2.3.1. (observation table) An observation table is a 3-tuple (S, E, T), with:

- S is a nonempty finite prefix-closed set of strings
- E is a nonempty finite suffix-closed set of strings
- T is a mapping $((S \cup S \cdot A) \cdot E) \rightarrow \mathbb{B}$, where $\mathbb{B}$ is $\{0, 1\}$

A set is prefix closed iff every prefix of every member of the set is also contained within the set, e.g.

$$\forall w \cdot a \in S : w \in S$$

Suffix closedness is defined analogously.

For two sets A and B the cartesian product is defined as $A \cdot B = \{(a, b) | a \in A, b \in B\}$

An observation table can be visualised as a two-dimensional array, where the rows are indexed by entries from $S \cup S \cdot A$, and the columns by E. The entries in the table are kept in T, which can be seen as a map that stores the results of membership queries. The entry indexed by $s \cdot e$ with $s \in S$ and $e \in E$ is denoted as $T(s \cdot e)$. We can also define a function $row(s)$ which retrieves a row from the observation table,

$$row(s) = [T(s \cdot e) | \forall e \in E]$$

		E		
		e_1	...	e_i
S	s_1	$T(s_1 \cdot e_1)$	...	$T(s_1 \cdot e_i)$
	...	...	...	...
	s_j	$T(s_j \cdot e_1)$	...	$T(s_j \cdot e_i)$
$S \cdot A$	$s_1 \cdot a_1$	$T(s_1 \cdot a_1 \cdot e_1)$	...	$T(s_1 \cdot a_1 \cdot e_i)$
	...	...	...	...
	$s_j \cdot a_k$	$T(s_j \cdot a_k \cdot e_1)$	...	$T(s_j \cdot a_k \cdot e_i)$

Table 2.1: Example observation table structure

It is convenient to separate the sections S and $S \cdot A$, since they represent different things. As we see later on, the unique rows indexed by S represent states in the the hypothesis represented by the observation table, and the one letter extension rows from $S \cdot A$ are used to determine the transitions in the hypothesis.

There are two important properties an observation table must have before it can be used to create a hypothesis: *closedness* and *consistency*.

Definition 2.3.2. (closedness) An observation table is closed if for each row($s \cdot a$) in $S \cdot A$, there exists a row(s) in S so that row(s) == row($s \cdot a$)

Definition 2.3.3. (consistency) An observation table is consistent if for each pair s_1 and s_2 both in S where row(s_1) == row(s_2), for all $a \in A$ row($s_1 \cdot a$) == row($s_2 \cdot a$).

When an observation table is both *closed* and *consistent*, it can be used to build a hypothesis in the form of a DFA $(Q, \Sigma, q_0, \delta, F)$, with:

- $Q = \{row(s) : s \in S\}$
- $\Sigma = A$
- $q_0 = row(\epsilon)$
- $\delta(row(s), a) = row(s \cdot a)$
- $F = \{row(s) : s \in S, T(s) = 1\}$

Learner

Now that we have introduced the concept of an observation table, we can specify the complete learning algorithm L*:

input : A leaner L and a teacher T
output: A hypothesis H matching the unknown system

$C \longleftarrow \epsilon$ // Keep track of counterexample C
$H \longleftarrow none$
$S \longleftarrow \{\epsilon\}$
$E \longleftarrow \{\epsilon\}$

while $C \neq none$ **do**
  while (S, E, T) *is not closed or not consistent* **do**
    if (S, E, T) *is not consistent* **then**
      find $s_1, s_2 \in S$, $a \in A$ and $e \in E$ such that:
      $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
      $E \longleftarrow E \cup \{a \cdot e\}$
      fill observation table using membership queries
    end
    if (S, E, T) *is not closed* **then**
      find $s_1 \in S$, $a \in A$ such that:
      $row(s_1 \cdot a)$ not in $\{row(s) | \forall s \in S\}$
      $S = S \cup s_1 \cdot a$
      fill observation table using membership queries
    end
  end

  /* at this point we have a closed, consistent observation table and can create a hypothesis */

  $H \longleftarrow createHypothesis(S, E, T)$
  $C \longleftarrow equivalenceQuery(H)$
  if $C \neq none$ **then**
    $S \longleftarrow S \cup \{C$ and all prefixes of $C\}$
    fill observation table using membership queries
  end
end
return H

Algorithm 2: L* learning algorithm

Example

As an example, lets assume we are trying to learn the regular set accepted by the DFA from Figure 2.1, which is also described by the regular expression b*a+b[ab]*. We know the alphabet $\Sigma = \{a, b\}$.

L* initializes it's observation table, setting $S = \{\epsilon\}$ and $E = \{\epsilon\}$, and enters it's main loop, asking membership queries to the teacher to fill all empty positions in the table until it is closed and consistent. After this, the observation table looks like this:

	ϵ
ϵ	0
a	0
b	0

Table 2.2: Initial observation table

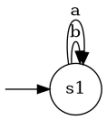

Figure 2.3: DFA H_1 consistent with the initial observation table

Since this observation table 2.2 is both closed and consistent, L* will use it to build a hypothesis DFA H_1

consistent with the observation table, which can be seen in Figure 2.3. After this, the L* learner poses an equivalence query to the teacher containing the hypothesis DFA. Since this DFA is obviously not equivalent to the "unknown" DFA $\mathbb{A}$ (2.1), a counterexample will be found. Let us assume the counterexample given back to the learner by the teacher is ab. We can see the hypothesis DFA output $\lambda_{H_1}(ab) = 0$, and $\lambda_{\mathbb{A}}(ab) = 1$, so ab and its prefix a are added to S. After this the new entries in the observation table are filled out using membership queries until it looks like the observation table in 2.3

	ϵ
ϵ	0
a	0
ab	1
aa	0
aba	1
abb	1
b	0

Table 2.3: Observation table after processing the counterexample ab

At this point, the observation table in 2.3 is closed, but not consistent since $row(\epsilon) = row(a)$ but not $row(\epsilon \cdot b) = row(a \cdot b)$, so b is added to E and the necessary membership queries to fill the new table are performed, resulting in the observation table in 2.4.

	ϵ	b
ϵ	0	0
a	0	1
ab	1	1
aa	0	1
aba	1	1
abb	1	1
b	0	0

Table 2.4: Observation table after adding b to E

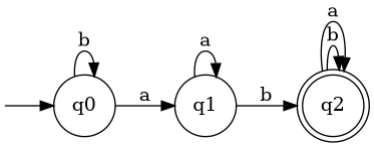

Figure 2.4: DFA H_2 consistent with observation table 2.4

After one last check if the table is closed and consistent, L* creates a new hypothesis H_2 and sends it to the teacher for an equivalence check. Since DFA H_2 is obviously equivalent to the target system, the teacher fails to find a counterexample and reports back to the learner that the hypothesis is equivalent. This causes L* to pose it's hypothesis H_2 as the final output, and the learning stops.

2.3.2. TTT

After looking at the tried-and-true L* algorithm, we should also introduce the current state of the art active learning algorithm, TTT [6]. TTT is also a learner in the MAT framework, but instead of the observation tables used by L*, it uses a **discrimination tree**.

This idea was first used by Kearns and Vazirani [26], and refined by Balcazar et al. [4] and Howar [20] and results in a massive reduction in the number of membership queries necessary to learn a model, since it reduces redundancy by only requiring membership queries that are necessary to distinguish states be performed. TTT is based on the "discrimination tree" algorithm by Howar, but further reduces redundancy by adding an extra step called discriminator finalization, which replaces excessively long suffixes in the discrimination tree with shorter ones to reduce the length of the membership queries asked.

Definition 2.3.4. (Discrimination Tree) A discrimination tree $\mathcal{T}$ is a rooted binary tree, where

- N is a set of nodes, which can be divided into inner nodes I and leaves L, e.g. $N = I \cup L$
- n_0 is the root node
- inner nodes are labeled with a discriminator (or distinguishing suffix)
- inner nodes have exactly two children, a $\top$-child and a $\bot$-child
- leaves are linked to hypothesis nodes

There are two main operations that are used to obtain information from a discrimination tree:

- sifting
- getting the lowest common ancestor of two nodes

Sifting is the main operation $sift(u)$ done on the discrimination tree. For a given access sequence s we can find out what node we end up in by sifting the access sequence through the discrimination tree. To describe this process informally: we start at the root node n_0, if it is a leaf, terminate and return the leaf. If it is not, append the discriminator v of the current discrimination tree node to the access sequence u and ask a membership query using $u \cdot v$. Depending on the answer of the membership query, take the edge to the corresponding child node and repeat this process until you end up in a leaf.

LCA, or lowest common ancestor, is the operation $lca(a,b)$ that retrieves the first inner node that is an ancestor to the two given leaf nodes a and b. Because of the way the discrimination tree is structured, this is always a distinguishing sequence for the two given nodes. If all nodes except the root node carry a pointer to their parent node, it can be computed very efficiently as well by simply walking up the tree towards the root node and returning the first node that is found in the path of both nodes towards the root.

Lets immediately dive into a concrete example of a DFA A, with it's corresponding discrimination tree D

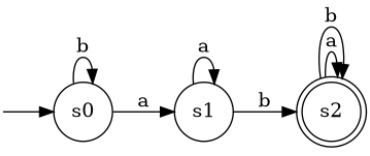

Figure 2.5: DFA A

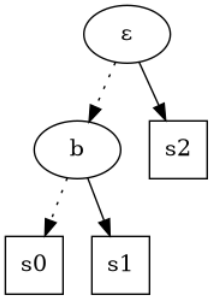

Figure 2.6: Discrimination tree D for DFA A

In the discrimination tree depicted in Figure 2.6, the circular nodes represent inner nodes, which are labeled with the discriminating suffixes. The square nodes represent leaf nodes, and are labeled according to which state in the hypothesis in figure 2.5 they belong to. Inner nodes have two children, to which the transitions are indicated with a dotted arrow for the $\bot$-child, and a solid arrow for the $\top$-child.

Sifting works as follows: Say we want to know what state we are in after running a certain sequence of inputs. For the sake of simplicity, let us use $u = \epsilon$. We start at the root of the tree which has the discriminator ϵ, so we do a membership query for $\epsilon \cdot \epsilon$, which returns false since it is not accepted by the target DFA A. This means we proceed along the $\perp$-path to the next inner node which is labeled with the discriminator b. We do another membership query, this time using the sequence $\epsilon \cdot b$, which also returns false, so we continue towards the $\perp$-child and end up in the leaf node labeled $s0$, so we know the sequence ϵ takes us to state $s0$

The lowest common ancestor operation kind of speaks for itself, but for the sake of completeness lets go through an example here too. Let's say we want to know how to distinguish state $s0$ and $s2$ from each other. We can find the discriminator that allows us to do this by finding the lowest common ancestor of the leaves labeled $s0$ and $s2$ in the discriminator tree. We simply walk back up towards the root from both nodes and return the first node that is in both paths, which in this case is the root node labeled ϵ. This means we can simply use the access sequences for nodes $s0$ and $s2$ and see what the output of a membership query is for them, since $\lambda(H\lfloor s0\rfloor \cdot \epsilon)$ is false, and $\lambda(H\lfloor s2\rfloor \cdot \epsilon)$ is true.

Counterexample decomposition

Another key element of TTT is the counterexample decomposition introduced by Rivest & Schapire [32], further elaborated by Steffen [38]. As we will see further on, as soon as TTT discovers a counterexample, it needs to decompose it in such a way that it can add the suffix of the counterexample to the discrimination tree, which is enough to distinguish between the hypothesis state that was wrong, and the new state introduced by the counterexample.

Rivest & Schapire state that for a given counterexample w between target DFA A and hypthesis H, there exists a decompostion $w = u \cdot a \cdot v$, with $u \in \Sigma^*$, $a \in \Sigma$, and $v \in \Sigma^*$, and $\lambda^A(H\lfloor u\rfloor \cdot a \cdot v) \neq \lambda^A(H\lfloor u \cdot a\rfloor \cdot v)$ This makes it apparent that, while in the hypothesis $H\lfloor u \cdot a\rfloor$ and $H\lfloor u\rfloor \cdot a$ lead to the same state, they have a different output for the separating sequence v: thus, this state in the hypothesis needs to be split.

Learner

Now that we introduced the concept of a discrimination tree, let us look at how the learning algorithm TTT works. It is significantly more complex than L*, and a 100% full formal description is outside of the scope of this thesis. The goal here is to instill a working knowledge of how TTT functions into the reader. For the full technical details, please refer to the PhD thesis of M. Isberner [21]

The TTT learner consists of three main datastructures: A spanning tree hypothesis Sp, which is a prefix-closed set of access sequences describing the path to all states in the hypothesis so far, A discrimination tree T, as described above, and a discrimination trie which is used to efficiently store the discriminators in a redundancy-free way. The discrimination trie is only used to improve memory efficiency, and has no effect on the amount of queries to be done, so for the sake of simplicity we can store the discriminators directly in the nodes of the discrimination tree and ignore the trie altogether.

In the original paper [22] , TTT is described as working in four main steps:

Hypothesis construction. To construct a hypothesis from it's internal datastructures, TTT looks at the prefix-closed set of access sequences $Sp \in \Sigma^*$. Each of these sequences is an access sequence to a state, so for each access sequence the hypothesis has a state. This hypothesis DFA is at first only connected by the spanning tree implicitly described in S. To get the non-spanning tree transitions, we need to sift them through the discrimination tree. To determine the a-transition for the state described by the access sequence $s \in Sp$, we sift the sequence $s \cdot a$ through the tree. The resulting leaf contains the target of the transition. We do this for all missing transitions in each state to get the complete hypothesis.

Hypothesis refinement Once the previous step has successfully constructed a hypothesis, it is passed to the teacher for an equivalence query. If no counterexample is found, the hypothesis is returned as the final answer. If a counterexample is found, it is decomposed into a prefix u, breaking point a, and suffix v. From this we can deduce that the sequences $H\lfloor u\rfloor \cdot a$ and $H\lfloor u \cdot a\rfloor$ should lead to different states, yet according to the current hypothesis they are the same. This means that this state in the hypothesis should be *split*. In other words, a new state with the access sequence $H\lfloor u\rfloor \cdot a$ should be introduced to Sp. Note that this preserves prefix closedness.

The discrimination tree also needs to be updated. The leaf linked to the original state in the hypothesis that was split in the previous step needs to be split as well. This is done by replacing it with an inner node labeled with the suffix v, which can discriminate the the new state and the old state, which are added to the new inner node as children according to their output after the suffix it is labeled with.

Hypothesis stabilization As a result of the hypothesis refinement step, it is possible for the hypothesis to become inconsistent with the discrimination tree. For example, it is possible for a state to be the $\perp$-child of an inner node in the discrimination tree, but still be an accepting state in the hypothesis. This means the access sequence for that state, plus the distinguishing suffix in the tree form an *internal* counterexample. This counterexample can be treated in the same way as a counterexample retrieved from the teacher, e.g. it can be decomposed, used to split a state in the hypothesis, and update the discrimination tree.

Discriminator finalization This is where TTT sets itself apart from the previous work in discrimination tree based learners. Discriminators added to the tree through counterexamples are marked *temporary*, and can contain a relatively long suffix v from the previously done decomposition. Discriminator finalization attempts to replace the root of a subtree of temporary discriminators (also called a block) with a shorter, non-temporary discriminator. This is done by either finding two states x and y in the same block that are themselves output-separable, e.g. $\exists a \in \Sigma^* : \lambda(\lfloor x \rfloor \cdot a) \neq \lambda(\lfloor y \rfloor \cdot a)$, or separable through their a-successors, e.g. the a-successors of state x and y are in distinct blocks, which implies they can be separated by a non-temporary discriminator in the tree. When two such states x and y can be found, the non-temporary discriminator separating them can also be found, which then in turn can be used to replace the inner node in the discrimination tree at the block root.

Note: this new discriminator that replaces the temporary one does not necessarily partition the subtree in the same way, so some effort is required to rearrange the subtree. This is done by marking every leaf in the subtree according to it's output relative to the new discriminator, and propagating this label up towards the root of the block. Then, for the $\perp$-subtree of the new final inner node, keep only the $\perp$-labeled nodes, and for the $\top$-subtree only the $\top$-labeled nodes. For a more formal description of this process please refer to [21].

these four steps above are essentially repeated in a loop, until no counterexample can be found anymore and the final hypothesis is returned.

Example

Lets walk through an example of TTT learning the DFA A 2.5 as in our previous examples. TTT starts by initializing it's prefix closed set of access sequences Sp with one member, ϵ, for the first state in the hypothesis. Consequently, the discrimination tree is initialized with a leaf as root node, which is linked to the initial state.

Now TTT takes the first step of building a hypothesis, which looks like Figure 2.7. The corresponding discrimination tree at this point in time looks like Figure 2.8, which as we can see consists of a single leaf which is linked to the only state in the hypothesis, $s0$.

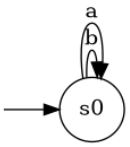

Figure 2.7: Initial hypothesis DFA H_0

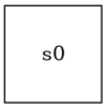

Figure 2.8: Initial discrimination tree

During the second step, TTT passes the initial hypothesis H_0 to the teacher to perform an equivalence check. This hypothesis is obviously not equivalent to the DFA A we are trying to learn, so a counterexample can be found. Let us assume the counterexample returned by the teacher is abb.

TTT then decomposes the counterexample into a prefix u, pivot point a and suffix v. The decomposition process looks for a decomposition of the counterexample where $\lfloor u \rfloor_{H_0} \cdot av$ gives a different output from $\lfloor ua \rfloor_{H_0} \cdot v$, which in this case is true for $u = \epsilon, a = a, v = bb$ since $\lambda_A(ab) \neq \lambda_A(abb)$.

Now that we have the decomposition, we know the state accessed by $\lfloor ua \rfloor_{H_0}$ needs to be split. This is the state s_0, so we look up the leaf in the decision tree which it is linked to, and split it. This is done by replacing the current node by a new inner node labeled with the suffix v, adding the old leaf node to it as a child, as well

as a new leaf node which will be linked to the new state created by the split. This state s_1, with access sequence $\lfloor u \rfloor_{H_0} \cdot a$, is also added to the prefix closed set Sp. After this process is finished, the resulting hypothesis H_1 and the corresponding decision tree look like the following:

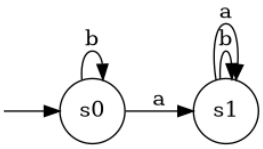

Figure 2.9: hypothesis DFA H_1

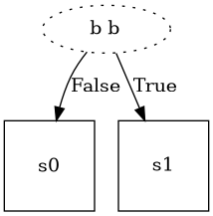

Figure 2.10: Discrimination tree after processing the counterexample abb

This hypothesis and discrimination tree are not *stable*, e.g. the discrimination tree is not consistent with the hypothesis. We can see this by sifting the access sequence a into the tree, which on the inner node labeled bb takes us along the "True" branch since $\lambda_A(abb) = 1$. This means that, according to the decision tree, the state with access sequence a should be an accepting state, yet this is not the case in the hypothesis, therefore abb forms an *internal counterexample*. An internal counterexample can be treated in the same way as a counterexample obtained from the teacher, so we go through the same process described above another time. We decompose the counterexample, this time into (ab, b, ϵ), and create a new state with access sequence ab, and split the corresponding leaf in the decision tree, resulting in the following:

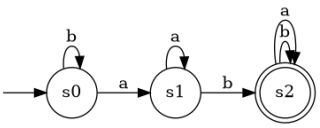

Figure 2.11: hypothesis DFA H_2

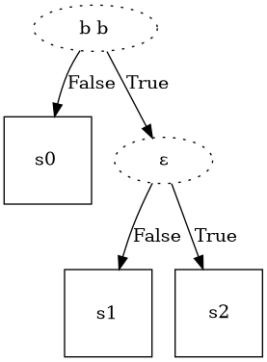

Figure 2.12: Discrimination tree after processing the internal counterexample abb

We now have a correct hypothesis H_2, as well as a stable decision tree. However, there is still one last step to go. In this example it may not be of much benefit since essentially we are already done, but when learning larger state machines *discriminator finalization* can gain us a significant improvement in symbol complexity (e.g. the amount of symbols sent to the SUL to execute). It is done as follows: First, we need to gather the *blocks* of the discrimination tree. A block is a maximal subtree of temporary nodes, so in this case the entire tree is one block. TTT then looks for a pair of states in the block that can either be separated by their output, or by their a-successors (see the section describing discriminator finalization 2.3.2). In this case TTT will find an output-wise splitter b, which is suited to replace the root node of the block bb.

Then, the new block with the root node marked ϵ can also be split output-wise using a as new final discriminator. Now, all temporary nodes have been replaced with final ones, resulting in the final discrimination tree seen in Figure 2.13.

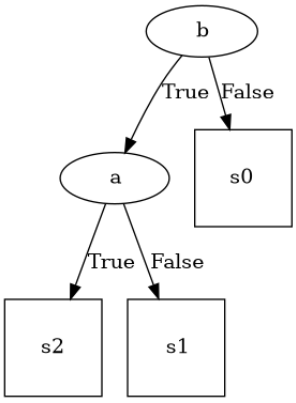

Figure 2.13: Finalized discrimination tree

2.3.3. Adaptation for Mealy machines

Both L* and TTT were originally designed to infer DFAs, but fortunately they can both be adapted to suit our purpose of learning Mealy machines relatively easily. L*'s observation table can be modified so that the mapping T: $((S \cup S \cdot A) \cdot E) \rightarrow \mathbb{B}$ changes to $((S \cup S \cdot A) \cdot E) \rightarrow \Omega$, where Ω is the output alphabet of the unknown Mealy machine. This effectively means that the cells in the observation table simply contain the output of the complete trace that indexes that cell. Now this observation table can be used to construct a Mealy machine in the same way as would normally be the case, and no further modifications are necessary.

Adapting TTT to learn Mealy machines is slightly more challenging. We need to change the discrimination tree from a binary tree to a tree where each node can have an arbitrary number of children, along with some other adaptations which are described in the PhD. Thesis of M. Isberner [21].

2.4. Equivalence oracles

Up until this point we have focused mostly on the learner-part of the MAT framework. However, without the ability to pose equivalence queries, the learner would not be able to infer a correct hypothesis. Until now we have assumed the existence of an equivalence oracle which is able to answer whether or not a given hypothesis is equal to the target system, and if not, return a counterexample. In practice no such oracle exists, so it is necessary to approximate equivalence queries using equivalence checking techniques. It is important to note that, while there are a lot of different methods available, they all have their strengths and weaknesses and provide different guarantees. Most of all they all merely approximate equivalence queries and thus can give an incorrect output, although usually only false positives, e.g. they answer that a hypothesis is equivalent while it is not. In any case, they all work by generating a set of test queries, which are executed on both the hypothesis and the SUL. If the hypothesis and the SUL give the same output for all the test queries, they are assumed to be equivalent. If the output is different for a certain query, that query is returned as a counterexample. In this section we will explain several of the most commonly used equivalence checking techniques and discuss their strengths and weaknesses.

2.4.1. Brute force

Perhaps the simplest equivalence checking technique is brute forcing through all possible input combinations up to a certain length n. While this is guaranteed to find a counterexample if one exists up to the specified length, it is perhaps the least efficient way of equivalence checking. The amount of queries in the test set grows exponentially with n and thus the query complexity is $\mathcal{O}|\Sigma|^n$, where $|\Sigma|$ is the size of the input alphabet. To generate the test set, we use the previously defined cartesian product and introduce a repeated variant of it for any set A:

$$A^n = \underbrace{A \cdot A \cdot \ldots \cdot A}_{\text{repeated n times}}$$

The algorithm for brute force equivalence checking goes as follows:

input : hypothesis H, SUL S, input alphabet Σ, max length n
output: Counterexample C if one exists, $None$ otherwise

for $i \leftarrow 1$ **to** n **do**
  $T \leftarrow \Sigma^i$
  for $t \in T$ **do**
    $SULOutput \leftarrow \lambda_{\text{sul}}(t)$
    $HypOutput \leftarrow \lambda_{\text{hyp}}(t)$
    if $SULOutput \neq HypOutput$ **then**
      return t
    end
  end
end
return $None$

Algorithm 3: Brute force equivalence checking algorithm

Because of the exponential complexity of the brute force equivalence checking algorithm, it's practical usefulness is limited. Anything but the smallest state machines would require n to be so large that running it quickly becomes infeasible, but for the sake of completeness it is still included here.

2.4.2. Random walk

While brute forcing equivalence queries guarantees finding a counterexample up to length n if one exists, it is too expensive to be feasible. Using a random walk to generate input sequences allows us to explore the same input space of sequences up to a given length n, but while covering more ground faster by not exhaustively searching the inputs of lower length before moving on to larger sequences. However, the random walk method does not guarantee finding a counterexample up to the specified length if one exists, because it isn't likely to explore everything in a realistic setting.

For convenience, we use λ^{step} as a step output function here, meaning it takes in a single symbol from Σ, takes the system to the next state, and gives the output of the transition, as opposed to the normal output function λ which takes in an entire sequence and gives the last single output for that sequence.

input : hypothesis H, SUL S, input alphabet Σ, max length n, number of walks $numwalks$
output: Counterexample w if one exists, $None$ otherwise

for $i \leftarrow 0$ **to** $numwalks$ **do**
  reset S to initial state
  reset H to initial state
  $w \leftarrow \epsilon$
  while *length of* $w \leq n$ **do**
    $a \leftarrow$ random element from Σ
    $w \leftarrow w \cdot a$
    $SULOutput \leftarrow \lambda^{step}_{\text{sul}}(a)$
    $HypOutput \leftarrow \lambda^{step}_{\text{hyp}}(a)$
    if $SULOutput \neq HypOutput$ **then**
      return w
    end
  end
end
return $None$

Algorithm 4: Random walk equivalence checking algorithm

2.4.3. W-method

A more efficient method of equivalence checking that still provides certain guarantees about finding counterexamples is the W-method, proposed by Chow in [10]. To provide these guarantees, the W-method needs an upper bound m for the amount of states in the unknown system. For a hypothesis consisting of n states, W-method creates a test suite by concatenating three sets of inputs, a *state cover set* V, *test cases* $Z = \prod_{i=0}^{m-n} \Sigma^i$, and a *characterizing set* W. Resulting in a test suite $V \cdot Z \cdot W$.

Definition 2.4.1. (State cover set) A state cover set V for a state machine S is a set of input traces that reaches every state in S. E.g. $\forall q \in Q_S : \lfloor q \rfloor_S \in V$. $V = \{\lfloor q \rfloor_S | \forall q \in Q_s\}$ It can be easily determined by performing a breadth first search on S and recording the inputs for each state when it is first reached.

Definition 2.4.2. (Characterizing set) A characterizing set W for a state machine S is a set of traces that can be executed on from any state $q \in Q_s$ which outputs uniquely identify that state q. E.g. the outputs of $q \cdot W$ are unique for all $\lfloor q \rfloor \in Q_s$. Determining the characterizing set for a given state machine S can be done in several ways. For example the set E in the observation table utilized by L* is a characterizing set. Other ways of determining the characterizing set are partition refinement like used in Moore's minimization algorithm [31] or Hopcrofts algorithm [19], or the more efficient method by Smetsers et al. in minimal separating sequences for all pairs of states [35].

A simple pseudocode example of the W-method could look as seen in Algorithm 5. In practice, it might be beneficial to generate the test traces "as you go", since the complete test set can become quite large and require a lot of memory to be stored up front completely.

input : hypothesis H, SUL S, input alphabet Σ, upper bound on SUL states m
output: Counterexample w if one exists, $None$ otherwise

$V \leftarrow statecover(H)$
$Z \leftarrow \prod_{i=0}^{m-n} \Sigma^i$
$W \leftarrow characterizingset(H)$

for $w \in V \cdot Z \cdot W$ **do**
  $SULOutput \leftarrow \lambda_{\text{sul}}(w)$
  $HypOutput \leftarrow \lambda_{\text{hyp}}(w)$
  if $SULOutput \neq HypOutput$ **then**
    return w
  end
end
return $None$

Algorithm 5: W-method equivalence checking algorithm

Let us look at an example of how the W-method can be used to approximate an equivalence query in the MAT framework. Assume at some point during the learning process, our learner has come up with the hypothesis in Figure 2.14, and is trying to learn the DFA in Figure 2.15. Assume we know the target DFA has 3 states, so we pass this as input to the W-method along with the hypothesis and target DFA.

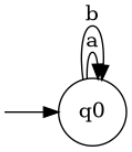

Figure 2.14: Example hypothesis

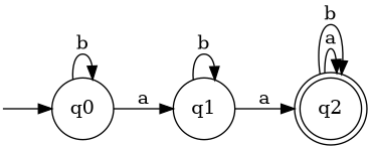

Figure 2.15: Target DFA

The W-method will then calculate the state cover set for the hypothesis, which in this case is simply $V = \{\epsilon\}$ since we do not need to input anything to get into the initial state, and there are no other states. Since the hypothesis has one state ($n = 1$), and we provided as input to the algorithm $m = 3$, the test sequences Z will become $\prod_{i=0}^{2} \Sigma^i = \{a, b, aa, ab, ba, bb\}$. The distinguishing set for our hypothesis is $W = \{\epsilon\}$ which satisfies the requirement of all states having a unique output for all traces in the set, since there is only one state in the hypothesis. Therefore the complete test set in this case is $V \cdot Z \cdot W = \{a, b, aa, ab, ba, bb\}$. These test queries will be posed to the SUL and a counterexample aa will be found since this input leads to an accepting state, which is not present in the hypothesis.

2.5. Characterizing sets

Like mentioned in definition 2.4.2, a characterizing set for a given state machine S is a set of input sequences of which the output uniquely identifies every state in S. Every minimal (e.g. no two states show identical behavior) state machine has a characterizing set. In the following section we will outline one of the possible methods to find a characterizing set. A newer, improved variant of this method is described in [35].

2.5.1. Partition refinement

One way to find a characterizing set for a given state machine S, is building a partition tree using a partition refinement algorithm. A partition tree can be built as a tree where each node contains a partition block holding states of S. The initial (trivial) partition contains a single block which contains all states, and this partition is repeatedly refined for each layer it goes down in the tree. The refinement process is repeated until all blocks in the partition only contain equivalent states, e.g. the output for all states in a block is equivalent for all inputs.

Two examples of algorithms using partition refinement are Moore's and Hopcroft's algorithms for DFA minimization. They use the idea of building a partition tree until no blocks can be refined anymore, which

means that if there are blocks in the tree that contain more than one state, that those states are necessarily equivalent. This fact can be used to minimize DFAs, since all distinguishable states correspond to a partition, so a new, minimal DFA can be constructed using the final partitioning as states.

While minimizing DFAs or other state machines is not what we are looking for, the way these algorithms build their partition tree is still useful for our purpose of determining a distinguishing set. We will end up with partition blocks containing only a single state, since in our case all states are unique, but we can use the set of "splitters" used to partition the states as a distinguishing set.

The paper Minimal Separating Sequences for All Pairs of States [35] provides a solid explanation of using a partition refinement algorithm to determine a distinguishing set for a given FSM, but we will summarize the procedure here. In essence, you start with a partition P containing the set Q of all states in your state machine, and iteratively refine P until all states in each block are equivalent. Like before, two states q_1 and q_2 are equivalent if $\lambda(q_1, x) = \lambda(q_2, x)\ \forall x \in \Sigma^*$. Analogously, a separating sequence for two states q_1 and q_2 (sometimes called splitter in the context of splitting a partition block) is any sequence $x \in \Sigma^*$ such that $\lambda(q_1, x) \neq \lambda(q_2, x)$.

To refine a partition, we look for a block to split, e.g. the block contains states that we can prove are inequivalent by giving them a certain input for which they will give a different output. We can split a block B for two reasons:

- we can find an input symbol a for which two states $q_1, q_2 \in B$ have a different output, $\lambda(q_1, a) \neq \lambda(q_2, a)$
- we can find an input symbol a that takes two states $q_1, q_2 \in B$ to new states $q_1' = \delta(q_1, a)$ and $q_2' = \delta(q_2, a)$ for which holds that q_1' and q_2' are in different blocks.

In the first case, we simply create a child node for each output given by the states after consuming symbol a, and add these as children of the node containing the original partition B. The second case is a little more complicated. Since we know that q_1' and q_2' are in different blocks, this means they have a *lowest common ancestor* node (see 2.3.2) which has a splitter a' that separates q_1' and q_2'. Since q_1' and q_2' were reached from q_1 and q_2 by consuming input a, we know the splitter for q_1 and q_2 can be created by prepending a to a'. We can use the output of the states in B for this sequence $a \cdot a'$ to divide them over the child nodes as in the previous case.

Before we can assemble our complete algorithm, we need to define two more concepts for partitions: **stability** and **acceptability**.

- A partition is **acceptable** if for all blocks, all states contained within each block have the same output for all single symbol inputs $a \in \Sigma^*$, e.g. for all pairs $(q_1, q_2) \in B$ and $\forall a \in \Sigma^*$, $\lambda(q_1, a) = \lambda(q_2, a)$.
- A partition is **stable** if it is acceptable and for any input $a \in \Sigma^*$ and all pairs of states $(q_1, q_2) \in B$, $q_1' = \delta(q_1, a)$ and $q_2' = \delta(q_2, a)$ are also in the same block.

The algorithm to find a partition tree for a given FSM M is as follows:

input : A FSM M
output: A valid and stable partition tree T

Initialize T with a single node with one block containing the trivial partition of M

while *P(T) is not acceptable* **do**
  find a so that $\lambda(q_1, a) \neq \lambda(q_2, a) for (q_1, q_2) \in B$
  add child nodes to the node containing B using a as described earlier
end
while *P(T) is not stable* **do**
  find a so $q_1' = \delta(q_1, a)$ and $q_2' = \delta(q_2, a)$ such that q_1' and q_2' are in different blocks.
  find $LCA(q_1', q_2')$ and use it's splitter a' to construct $a \cdot a'$
  add child nodes to the node containing B using $a \cdot a'$ as described earlier
end
return T

Algorithm 6: Partition refinement algorithm, adapted from Smetsers et al. [35]

2.6. Combining active and passive learning

In the work by N. Yang et al. [41], the tradeoff between learning time and completeness of learned models was investigated with a pilot study done at ASML. To resolve the tradeoff, they combined active learning techniques with passive learning results as well as execution logs containing traces of the running programs during normal use.

One observation they made is that for systems where early choices affect behavior much later in the program, it becomes challenging to learn the full behavior of that problem using active learning alone. This makes sense since commonly used equivalence checking techniques like the W-method check behavior up until a certain depth, so if a choice made 10 actions ago influences the outcome of the next step in a system, this will not be seen by the equivalence checker if it has a depth lower than 10. Fortunately, they found that this behavior is often found during normal use of the program and thus can be seen in the execution logs or models derived from them by for example passive learning.

To summarize the method used in this paper, they define three oracles (which could be seen as three separate equivalence checkers):

- A log-based oracle, which attempts to find counterexamples by identifying traces present in the log but which cannot be generated by the current hypothesis being checked. They do this by building a prefix tree acceptor (PTA) from the logs, and calculating a difference automaton between the PTA and the hypothesis.
- A Passive learning based oracle, which works similarly to the log-based oracle but instead computes a difference automaton between a model learned from the logs using passive learning and the hypothesis. They note that this oracle is based on a conjecture that passive learning overgeneralizes, and so they need to check any counterexamples found this way by running them on the actual system to see if it is a real counterexample, or a mistake made by overgeneralization by the passive learning algorithm.
- A wp-method based oracle. This is a standard equivalence checking technique that like the W-method provides certain guarantees about finding a counterexample if one exists up to a certain depth.

However, since the standard MAT-framework is not built to use multiple equivalence oracles, they define a sequential equivalence oracle which contains the other three and executes them in order. When an equivalence query is posed, it is first passed to the fastest oracle, the log-based oracle. If no counterexample is found in the logs, the passive learning oracle attempts to find one. Finally if passive learning also does not find a counterexample, a standard equivalence check is ran using the wp-method. The authors mention that this idea is similar to the hierarchical memory architecture used in computers, with different levels of caches of different sizes and speeds. If checking the fastest, smallest cache does not result in a hit, go one level up and try the slightly slower, larger cache. If that also does not result in a hit, go up another level etc. A description of this architecture can be seen in Figure 2.16

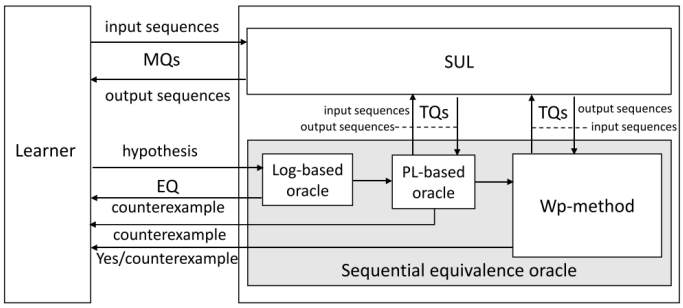

Figure 2.16: Sequential oracle architecture, adapted from N. Yang et al. [41]

To test whether or not this approach was helpful, they ran experiments on 18 different model-driven software engineering based components used in ASML's lithography machines. They conclude that both the full setup of the sequential equivalence oracle, as well as two simplified variants using only the log-based and passive learning based oracles significantly speed up the learning process. However, there was no siginificant benefit over using both at the same time.

Main takeaway

For the purposes of this thesis, the most interesting part of this work is the concept of the sequential equivalence oracle. In the following chapters we will introduce new equivalence checking methods intended to speed up the active learning process much like the log-based and passive learning based oracle in this paper do. While they are much faster than a standard W-method check, they do not provide much in the way of completeness or under/over-approximation guarantees. As such they need to be backed up by the W-method or similar much like the last step in the serial equivalence oracle here.

2.7. Combining learning and fuzzing

In the master thesis of Mark Janssen [24] the idea of combining active learning algorithms with fuzzing is explored. In this work, a prototype tool is introduced that leverages the test cases found by a fuzzer (in this case AFL) and uses those to perform equivalence checks during active learning. The results of this method are compared to the L* and TTT implementations in LearnLib combined with the W-method, and the results are promising.

The implementation of this fuzzing equivalence oracle reads test cases from a fuzzer corpus, and uses these as input traces for an equivalence check. The author evaluates the performance of this fuzzing equivalence oracle on the RERS 2015 reachability problems. This is done by first fuzzing the problems for 24-48 hours using AFL, and then using the resulting fuzzing corpora for the equivalence checks during the active learning process. The results can be seen in Table 2.5.

Target	Method	States	Reachability	Time	Queries
Problem 1	TTT, W-method 1	25	19/29	00:00:04	7342
Problem 1	L*, W-method 8	25	19/29	13:49:10	246 093 350
Problem 1	TTT, fuzzing eq.	334	29/29	00:00:21	16 731
Problem 1	L*, fuzzing eq.	1027	29/29	00:44:40	2 860 990
Problem 2	TTT, W-method 1	188	15/30	01:00:05	8 156 730
Problem 2	L*, W-method 3	195	15/30	17:05:59	239 851 191
Problem 2	TTT, fuzzing eq.	2985	24/30	00:13:06	412 340
Problem 2	L*, fuzzing eq.	3281	24/30	13:28:42	42 120 554
Problem 3	L*, W-method 1	798	16/32	14:15:36	1 421 330 223
Problem 3	TTT, fuzzing eq.	1054	19/32	00:13:27	698 409
Problem 3	L*, fuzzing eq.	1094	19/32	13:28:42	23 464 145
Problem 4	TTT, W-method 7	21	1/23	04:11:13	51 760 913
Problem 4	L*, W-method 7	21	1/23	03:10:51	51 759 741
Problem 4	TTT, fuzzing eq.	7402	21/23	00:16:48	458 763
Problem 5	L*, W-method 1	183	15/30	00:13:17	2 203 813
Problem 5	TTT, fuzzing eq.	3376	24/30	00:08:00	416 943
Problem 6	L*, W-method 1	671	16/32	21:33:10	889 677 067
Problem 6	TTT, fuzzing eq.	3909	23/32	00:45:00	1 804 595

Table 2.5: Results of combining learning and fuzzing on RERS2015 reachability, adapted from M. Janssen [24]

On the reachability problems of RERS 2015, the proposed method outperforms the plain L* and TTT combined with W-method equivalence checks by a large margin. Where the non-fuzzing learning process fails to find many states or fail to finish the learning process in a reasonable time, the fuzzer based process reaches more error states, usually in less time. It is also stated that on the larger problems, running L* does not finish within multiple days of running and thus is excluded. Note that even after 24-48 hours of fuzzing, still not all reachable error states have been found by AFL.

In chapter 5, we dive deeper into leveraging fuzzers for active learning. We show that using only the traces generated by a fuzzer for equivalence checking is not always enough to capture the full behavior of a program, and try to resolve this by stacking the fuzzing equivalence checker with conventional methods to not break the guarantees provided by them.

Main takeaway

The use of fuzzing for active learning is a promising way to speed up the equivalence checking step, and it manages to find more states and reach more errors than the traditionally used W-method in less time. However, it does not provide similar guarantees, so caution is advised in case fully complete models are required.

3

Automata learning in practice

3.1. An initial look at RERS

After looking at the various techniques available for active state machine learning in chapter 2, let us take a look at how they can be used in practice. To do this, we will introduce the Rigorous Examination of Reactive Systems challenge. In the RERS challenge, participants are invited to solve various problems in two categories: linear temporal logic, and reachability. In this work, we apply our state machine learning library to both categories to help highlight it's flexibility. As we will see, learning state machines is very useful for the linear temporal logic problems and performs extremely well in this track. The reachability problems are a good benchmark to see how far we can push active state machine learning, but since it is a fully black box technique it is very challenging to compete with white/greybox techniques like fuzzing.

RERS allows participants a lot of freedom in how they solve the tasks. RERS 2019 saw participants use tools like AFL, LLVM Interval analysis, and various other analysis and verification tools. There are no restrictions on time or resource usage, and combining different tools is encouraged.

In this work, the RERS code is interacted with as if it were a Mealy machine. This is an excellent fit, since the problem code works by taking in input, which changes the internal state and gives an output depending on what state it is in. Obviously we do not know what the final Mealy machine should look like, but some training examples are provided which include a list of reachable error states. This allows us to check if our method is working correctly. An example state machine learned using L* on RERS problem 11, which is the simplest training problem, can be seen in Figure 3.1.

Figure 3.1: Mealy machine of RERS Problem11

Linear Temporal Logic

In the linear temporal logic (LTL) track, participants are given obfuscated code representing a reactive system, and are given a list of LTL propositions to check if they hold or not. The RERS LTL formulae consist of atomic propositions representing input and output symbols, as well as the following temporal operators[1]:

- $\boldsymbol{X\phi}$ (next): ϕ has to hold after the next step
- $\boldsymbol{F\phi}$ (eventually): ϕ has to hold at some point in the future (or now)
- $\boldsymbol{G\phi}$ (globally): ϕ has to hold always (including now)
- $\boldsymbol{\phi U\psi}$ (until): ϕ has to hold until ψ holds (which eventually occurs)
- $\boldsymbol{\phi WU\psi}$ (weak until, sometimes just **W**): ϕ has to hold until ψ holds (which does not necessarily occur)
- $\boldsymbol{\phi R\psi}$ (release): ϕ has to hold until ψ held in the previous step

When checking an LTL formula for a given system, the question is essentially: "can this system generate an input/output trace that violates the given formula?". For example, say we have a formula ($i\boldsymbol{U}o3$), and a system that generates an input/output trace $i, o1, i, o2, i, o3$. According to the above rules, the symbol U means "until", and thus the complete rule means: the trace only contains i until the symbol $o3$ occurs, which does eventually occur. As we can see in the trace, in the second step i does not hold anymore, and $o3$ has not occurred, so the formula is violated.

[1] http://rers-challenge.org/2020/index.php?page=problemDescS

Reachability

The goal of the reachability challenges is to discover what error states are reachable in given obfuscated code, which is available in both java and C. Some snippets of example code can be seen in listing 3.1. This is only a small exerpt from the code, the larger problems contain over 500,000 lines with many functions chained like this, so it would be totally infeasible for a human to make sense of them.

```
void calculate_outputm3(int input) {
    if((a1880323056 == 32 &&  cf==1 )) {
      calculate_outputm34(input);
    }
}
 void calculate_outputm49(int input) {
    if((a535900205 == 4 && (a1129397725 == 32 && (a856341416 == 33 && ((input == 9) &&  cf==1 ))))) {
      cf = 0;
      a363016538 = 34 ;
      a856341416 = 35 ;
      a1975051863 = 35 ;
       printf("%d\n", 25); fflush(stdout);
    } if((a1129397725 == 32 && (a856341416 == 33 && (( cf==1  && a535900205 == 4) && (input == 6))))) {
      cf = 0;
      a535900205 = 10;
      a1837354890 = 17;
       printf("%d\n", 18); fflush(stdout);
    }
}
```

Listing 3.1: Example RERS code

```

 if(((a1129397725 == 33 && a535900205 == 4) && a856341416 == 33)){
    cf = 0;
    __VERIFIER_error(0);
}
if(((a2099614420 == 7 && a535900205 == 9) && a856341416 == 33)){
    cf = 0;
    __VERIFIER_error(1);
}
if(((a1618049448 == 34 && a535900205 == 7) && a856341416 == 33)){
    cf = 0;
    __VERIFIER_error(2);
}
```

Listing 3.2: Example RERS error checking code

The main thing to figure out is which of the verifier errors can be triggered and which not. The RERS programs check for error conditions as shown in listing 3.2 By learning a state machine representing the RERS problems we can see which errors are reachable and which input sequence leads to them. An important caveat is that we cannot prove the absence of errors this way, since the learned state machine may not capture the entire system.

3.2. Practical considerations

Next to all the algorithms and theory, the design and implementation of the automata learning system can also greatly influence the time it takes to successfully learn a state machine. As we will see in this section, it is highly advisable to take a thorough look at ways to improve the speed at which queries can be asked. While this engineering work may not be the most academically interesting thing to do nor will it improve the computational complexity of the algorithms used, it can still take the runtime of certain learning processes from hours to minutes. Not to mention, while experimenting during development, it is beneficial to have a short feedback loop and to be able to quickly iterate instead of waiting for hours for experiments to run.

3.2.1. Query speed

Pexpect

It is possible to communicate with the RERS problems over stdin/stdout, so a natural first choice would be using a library such as Pexpect[2]. Pexpect is a python library that allows the user to open a subprocess, and essentially pretends to be a human typing commands and reading the corresponding output. This makes it very easy to automate interactive applications such as ssh, ftp, passwd and telnet. However, since it is designed for these kinds of tasks and not for high speed communications, it is rather slow.

Popen

In an attempt to speed up the process of asking queries, it was decided to do away with Pexpect and roll our own implementation, using python's built in Popen functionality. This allows hooking up a pipe to the stdin/stdout of the subprocess and directly reading from and writing to it. We implemented this as a connector that starts a separate thread for the communication with the subprocess so the main thread does not have to wait for the communication to finish, and a queue is used to communicate between the threads. While this does improve the query speed over Pexpect, profiling the application still showed the querying to be a bottleneck.

In-process querying

After optimizing the query speed over stdin/stdout as far as it would go, we drew some inspiration from two well known fuzzers, AFL and libfuzzer. In a way, fuzzing a process is similar to what we are doing here, in the sense that we need to rapidly try a huge amount of inputs, or in our case queries. At this point, we had been communicating over stdin/stdout, much like AFL does when fuzzing a process. However, libfuzzer takes a different approach. It is implemented as part of LLVM, and if a certain flag is used while compiling your program, LLVM essentially turns it into a fuzzer which is included in the same process. This means you can fuzz in-process, instead of having to go through the kernel for communication, which can be much faster and also scales better for multithreading.

This inspired us to write a patcher for the RERS programs to add a function to the code that allows us to ask a query by passing it as an argument to said function. This allows us to circumvent the scanf call that normally reads the input, thus removing the need for communication between processes. The RERS code is then compiled as a shared library, and loaded into the python process using python ctypes[3], which allows you to call C code from python. A small wrapper was written to allow the rest of the learning system to talk to it, and so the entire need for inter process communication is removed since only a single process remains.

3.2.2. Reset speed

Another thing preventing queries from being posed as quickly as possible is the fact that the system needs to be reset to get it back into it's initial state after every query posed. The RERS code does not contain a reset function or any other mechanism that would allow this to be done quickly, so it is necessary to kill the entire process and restart it to get a clean slate for the next query. Launching a new process is relatively expensive, especially if it needs to be done often like in our case. Therefore, creating a different method to reset the RERS programs can give us a significant speedup. This is done by writing another patcher that collects all the global variables in the RERS code, splits them into their definition and their assignments, and moves the assignments into a separate reset function. The main loop is also patched to add a special character which is not in the original alphabet, which triggers a reset. Using this technique we can perform a reset without having to tear down and relaunch the process.

[2] https://pexpect.readthedocs.io/en/stable/
[3] https://docs.python.org/3/library/ctypes.html

3.2.3. Comparison

Over the course of this thesis, all above techniques have been implemented in our automata learning library. Intuitively, each technique improves upon the next, so we verify this by comparing the query speed for all of them using different query lengths, averaged over 100 queries per length per technique. The results of this comparison can be seen in figure 3.2.

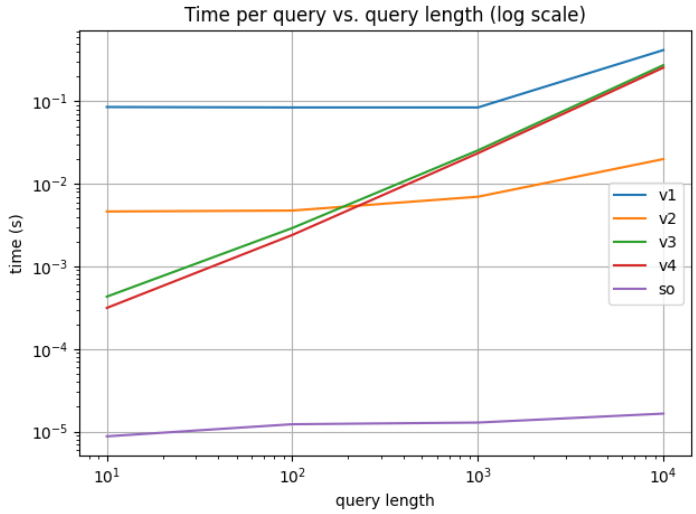

Figure 3.2: The time per query, for the different communication techniques

The legend in figure 3.2 indicates which type of connector to the RERS program was used. The labels have the following meaning:

- v1: PExpect based connector
- v2: Custom Popen based connector
- v3: Custom Popen with fast reset and built-in cache
- v4: Custom Popen with fast reset, no built-in cache
- so: In-process connector using RERS code compiled as a shared object

These results support the intuition we have about the speeds of the varying techniques, with the in-process communicator being the fastest of them all by several orders of magnitude. This was to be expected, as we can see a similar effect in other fields such as fuzzing, where similar techniques are used. We can therefore confidently recommend using this type of communication whenever possible (e.g. source code is available), as the only downside is the small amount of extra work required to write the extra function to call from the python side. One thing initially not expected however is the fact that the performance of the connectors using the fast reset technique degrades significantly faster with query length. This may be because of a sub-optimal implementation, but it was decided to not spend any more time on investigating this since the in-process connector outperforms all others by a large margin anyways.

3.2.4. Caching

Since posing queries is relatively expensive, it can be worthwhile to cache them. A naive approach would be to keep a map that stores the queries as keys, and their responses as values. This works, and in our implementation is the cache with the fastest lookup and insertion speeds given that these are constant time operations, but it consumes the most memory as no deduplication is performed whatsoever.

A more memory friendly approach would be to store the queries and their responses in a trie datastructure, which deduplicates the stored keys by building a tree like structure of nodes containing single elements from Σ. However, lookup and insertion are slower with time complexity of $O(m)$, where m is the lenght of the query.

To compare the two cache types, we ran 1000 queries of length 100 on RERS 2019 training problem 11 and recorded the time it took for the queries to run the first time, and how long it took to ask them the second time. The results, averaged over 10 runs, can be seen in table 3.1. These tests were all done using the v4 RERS connector. Memory use was determined by serializing the cache objects to a pickle string and recording the size of this string.

	Insertion time (s)	Retrieval time (s)	Memory use (bytes)
Dict	2.30942	0.00111	4554891
Trie	2.48066	0.03759	242983
Uncached	NA	2.30237	NA

Table 3.1: Insertion and retrieval times for different cache types

These results confirm our intuition, the dict based cache is the fastest, with the lowest insertion and retrieval overhead, while it uses the most memory. The Trie based cache has more insertion overhead, as well as the retrieval being significantly slower. However, as expected, it is much more conservative in it's memory usage.

In practice, for the larger RERS problems the dict cache uses too much memory, and the trie cache lookup is actually slower than the fastest connector described in the previous section, so in the final version of the RERS learning code no cache was used.

3.3. Low hanging fruit

Some relatively simple things can be done to speed up the process of learning state machines by applying some domain knowledge and applying smart heuristics. For example, in some cases it is possible to greatly increase the speed of the W-method by applying early stopping on certain outputs. In other cases, it may be beneficial to influence the order of access sequences by keeping track of certain statistics.

3.3.1. Early stopping

The RERS problems have alphabet sizes of between 10-20 characters, but for any given state usually not all of them are valid inputs. When an invalid input is given, the RERS program returns "invalid input" and does not transition to a different state.

Take for example the Mealy machine in Figure 3.3. Say we are performing an equivalence check using the W-method (see 5). In the standard implementation all sequences starting with b will be posed as a query just as the sequences starting with a. However, we know that transitions with the "invalid output" output keep the system in the exact same state, so we can safely ignore the whole branch of the testing tree starting with b.

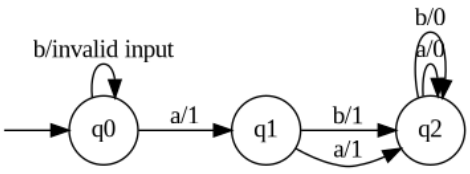

Figure 3.3: Example Mealy machine with invalid input

In this small example this might not seem like a huge difference in the amount of queries we need to do, but it still cuts the amount of queries necessary to check the state $q0$ in half. The larger the alphabet and the more invalid transitions are present, the greater the reduction in queries. In practice, some of the RERS problems are problematic to check beyond a depth of 3 or 4. With early stopping, W-method checks with a depth of 14 become tolerable time wise. Keep in mind that the standard W-method's test case query complexity is $O(|\Sigma|^D)$ where D is the depth.

Concretely, say we have an alphabet size $|\Sigma| = 10$, and each state on average has two valid inputs. This would mean that we could check a depth of 10 using early stopping in the same time as the standard W-method would need for a depth of 3, since $2^{10} \approx 10^3$.

3.3.2. Access sequence order

Another thing we can do to reduce the time needed to find counterexamples using equivalence checking is by applying heuristics to determine the order in which the access sequences are tried by the W-method. Recall that the W-method (2.4.3) takes all access sequences (the state cover set V) and concatenates them with the test sequences and distinguishing set to create the full test set. However, the algorithm does not specify in which order the access sequences should be used, and a naive implementation would most likely ask them in the order they are found, e.g. shortest first. As a result, the oldest states always get checked first, which may not be a very efficient way of working especially as the hypothesis grows larger and larger, since intuitively the older states in the hypothesis do not change as much as the newer ones since they have already been checked before.

In our library, several strategies are available:

- **shortest first** - test the states with the shortest access sequences first
- **longest first** - test the states with the longest access sequences first
- **interleave** - looping over all states, for each state ask all currently valid one-letter extensions before going to the next state
- **most interesting first** - states get a score according to how many times a counterexample was found by testing said state. States are tested with the highest scoring states going first. As soon as a state is completely checked and no counterexample was found, it's score gets reset.

Another interesting method of doing this could be by implementing power schedules, like used in fuzzers [5]. A power schedule is similar to the "most interesting first" strategy mentioned above, but works in a more refined way. In fuzzing with power schedules, seeds are assigned an "energy" value, which determines how often they are chosen to mutate and test on the system. The energy value of a seed can depend on multiple variables, like the execution time, length, and how many times the seed has lead to the discovery of new states. It seems worthwhile to explore the possibility of applying this idea to active state machine learning by assigning access sequences an energy value, and determining the order they are checked based on that.

3.3.3. Equivalence query approximation

By putting the techniques discussed above together, along with one more small adaptation we will describe in a minute, we can adapt the W-method equivalence checking technique and create a version of it that is better suited to the RERS challenge. Since the RERS challenge does not provide us with an upper bound of the amount of states in the problem we are trying to learn, we remove the upper bound on the amount of states in the SUL, and replace it with a search depth, or "horizon" parameter h. Simply put, this directly replaces the $m-n$ in $\prod_{i=0}^{m-n} \Sigma^i$, so the test cases of the W-method become $Z = \prod_{i=0}^{h} \Sigma^i$. This does break the guarantee of finding a counterexample if there exists one, but this is a practical tradeoff we have no choice but to make.

To efficiently implement early stopping, we need a way to store what paths lead to invalid transitions. We could store all input/output sequences that include "invalid input" as output in a trie and check if the given input crosses a node that is tagged with "invalid output", but this would take a lot of memory so we need something better. A similar, but more efficient implementation uses a **prefix set**.

Definition 3.3.1. (Prefix set) A prefix set P is a set that is said to contain an element e if a prefix of e is stored in P. For example, take $P = \{aa, abbbb\}$. this would mean $aaaaaa \in P$, but not $baaaaa \in P$. The addition of an element to a prefix set can remove elements from the set if it is a prefix of said elements. For example, adding ab to P would result in $P = \{aa, ab\}$ since ab is a prefix of $abbbb$.

In practice, a prefix set is implemented with the underlying data structure being a trie. This in combination with the fact that only the shortest prefix needed to accurately determine if an element is in the set is stored, makes a prefix set a good fit for our purpose of checking if an input sequence contains an invalid input.

With all that out of the way, we can define our "smart" W-method equivalence checking as follows in 7. Note that we explicitly reset the SUL and hypothesis to be able to get the intermediate output from the SUL without having to reset it twice, which would be inefficient. Also, directly implementing this pseudocode as is would be problematic for the "most interesting" ordering variant, since it requires keeping track of some

additional information that would have distracted from the main algorithm had it been included. However, since this is just bookkeeping and not very complicated it is left as an exercise for the reader.

input : hypothesis H, SUL S, input alphabet Σ, horizon h, outputs which to stop early on E, access sequence ordering $Order$
output: Counterexample w if one exists, $None$ otherwise

$V \leftarrow statecover(H)$
$W \leftarrow characterizingset(H)$
$P \leftarrow initializeprefixset()$

for $v \in Order(V)$ **do**
  $to\_visit \leftarrow \{a | \forall a \in \Sigma \text{ if } v \cdot a \notin P\} + \{\epsilon\}$
  while $|to\_visit| > 0$ **do**
    $cur\_test\_seq \leftarrow popleft(to\_visit)$
    `/* Grow the testing tree where no early-stop condition has been reached */`
    $SULOutputPre \leftarrow \lambda_{\text{sul}}(v \cdot cur\_test\_seq)$
    if $SULOutputPre \in E$ **then**
      add $v \cdot cur\_test\_seq$ to P
    else if $|cur\_test\_seq| \leq h$ **then**
      for $a \in \Sigma$ **do**
        if $v \cdot cur\_test\_seq \cdot a \notin P$ **then**
          add $cur\_test\_seq \cdot a$ to to_visit
        end
      end
    end
    `/* Perform the usual W-method tests */`
    for $w \in W$ **do**
      $SULOutput \leftarrow \lambda_{\text{sul}}(v \cdot cur\_test\_seq \cdot w)$
      $HypOutput \leftarrow \lambda_{\text{hyp}}(v \cdot cur\_test\_seq \cdot w)$
      if $SULOutput \neq HypOutput$ **then**
        return w
      end
    end
  end
end
return $None$

Algorithm 7: W-method equivalence checking algorithm with early stopping and ordering

In order to show the effectiveness of the modified W-method described in Algorithm 7, we learned state machines for the sequential training problems of RERS 2019, and logged the amount of queries done and the time taken. Both the smart and standard W-method were configured with a horizon of 3 and later 9, and the smart W-method was configured to stop early on invalid inputs and errors and used the "most interesting first" ordering strategy.

	type / depth	Total time (ms)	Test queries	States found	Errors found	Speedup
Problem 11	Standard 3	4783	627660	61	24	1x
	Smart 3	448	19542	61	24	10.67x
Problem 12	Standard 3	3555	395330	50	23	1x
	Smart 3	310	4758	50	23	11.46x
Problem 13	Standard 3	11154	1120801	59	17	1x
	Smart 3	755	52692	59	17	14.77x
Problem 11	Standard 9	420458	53566393	61	24	1x
	Smart 9	42782	2832398	61	24	9.82x
Problem 12	Standard 9	timeout	*437380471*	*50*	23	*NA*
	Smart 9	timeout	*101984919*	*279*	*26*	*NA*
Problem 13	Standard 9	timeout	*298424622*	59	17	*NA*
	Smart 9	66385	*4631657*	*59*	*17*	*NA*

Table 3.2: Standard vs smart W-method implementation

We see at least an order of magnitude reduction in the number of test queries (membership queries done during equivalence checking) and a corresponding reduction in the time required to learn the complete state machine.

Furthermore, for all runs that were able to run to completion, the amount of states found and errors reached remains the same, which indicates that we are only skipping queries that are indeed of no importance and correctness does not suffer.

For depth 9, the results are harder to interpret because of the timeouts but we can still see a similar reduction in both queries and time for problem 11. Problem 12 and 13 did not finish within an hour, so the learning was stopped at the one hour mark and statistics were recorded. For problem 12 we can see that the smart W-method was able to find more states and more errors with only a fourth of the amount of queries done by the standard W-method. For problem 13, the standard W-method was not able to finish within an hour, while the smart W-method was.

4

Exploiting structure in counterexamples to speed up equivalence checking

In the previous chapter we discussed using active state machine learning on the reachability problems, and how they have trouble competing with non-blackbox techniques. While dealing with the RERS 2019 training problem 12, we tracked the counterexamples found during the learning process, and noticed a certain pattern. Some of the found counterexamples can be seen below.

```
('7', '7', '7', '3', '1', '7', '7', '7', '7', '7', '1', '5')
('7', '7', '7', '3', '3', '10', '1', '7', '7', '7', '7', '7', '1', '5')
('7', '7', '7', '7', '7', '3', '1', '7', '7', '7', '1', '5')
('7', '7', '7', '7', '7', '3', '3', '10', '1', '7', '7', '7', '1', '5')
('7', '7', '7', '7', '7', '7', '7', '3', '1', '7', '1', '5')
('7', '7', '7', '7', '7', '7', '7', '3', '3', '10', '7', '1', '5')
('7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '7', '1', '5')
```

The corresponding state machine early on in the learning process can be seen in figure 4.1. We can clearly see some kind of repeating structure. Problem 12 is marked as an arithmetic problem, so perhaps the underlying code is counting something before a certain action can be taken. As we know, the code is heavily obfuscated so we can only guess.

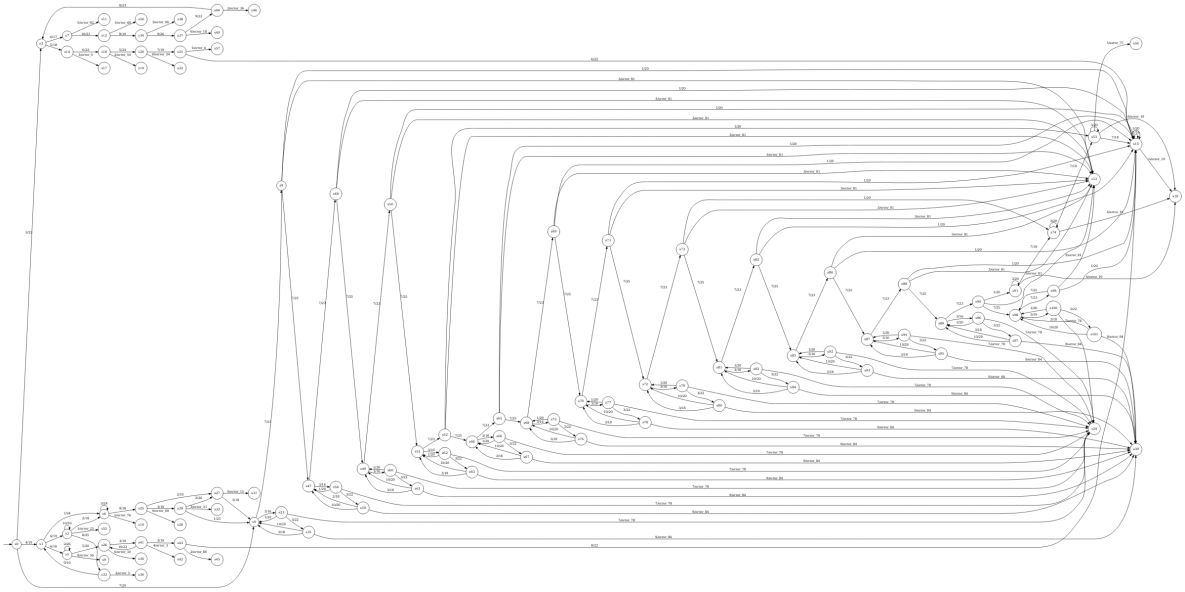

Figure 4.1: Incomplete state machine of training problem 12

One problem we encountered learning this state machine is that the W-method, even with early stopping and clever access sequence scheduling, takes a long time to go through all these repeating looping structures. This issue is compounded by the fact that the state machine learned from this problem easily consists of more than 3000 states, at which point the classic L* algorithm runs into severe difficulty since the observation table blows up in size, although TTT fares better. The downside of using TTT here is that it requires more equivalence queries than L*, and with the larger state machines those quickly get prohibitively expensive as well. To alleviate this problem, we should attempt to reduce the time-to-counterexample for each equivalence check as much as possible.

Since there is some kind of pattern in the counterexamples we can find for this problem, we theorized that we would be able to exploit this pattern to speed up equivalence checking. In essence, we would like to generate new sequences that look like the sequences we already have. Initially, we attempted to learn a regular language from the counterexamples that were found during normal equivalence checking, but this idea quickly ran out of steam since learning regular languages from only positive examples is problematic since most learning algorithms easily overgeneralize without negative examples [15]. This can be alleviated with clever use of statistics, but this seemed a very complex way to approach this problem so it was decided to try some simpler methods first.

4.1. Markov chain

Perhaps one of the simplest ways of generating new sequences that follow a similar pattern to a given set of sequences is using a Markov chain. A Markov chain is a stochastic model that can be used to predict the next element in a sequence. It is usually modeled as a state machine that transitions to other states with a certain probability, and this probability can only depend on the current state it is in (this is also called the Markov property). In NLP, Markov chains have been used as language models to generate new text similar to the body of text they were trained on. We can adapt this relatively simple method for our purposes and train a model on our counterexamples, and ask it to generate new ones.

4.1.1. Method

In this method, we train a Markov chain on a collection of counterexample traces and use it to generate potential new counterexamples. We train the Markov chain by splitting the input sequences into n-grams, recording the start n-gram for each sequence, and counting the occurrences of the next character in the sequence after a given n-gram for all n-grams in each sequence. Using this information, we can calculate the probability distribution of the start n-grams, as well as the probability distribution over the next characters for all n-grams. This gives us all the information we need to generate new sequences, which we do by picking a start n-gram according to their probabilities, and picking a next character according to the calculated probability distribution until a sequence of the desired length is obtained. If an unseen n-gram is reached, we pick a random character from the input alphabet as the next sequence element.

To use the Markov chain for equivalence checking, we train the Markov chain on the currently seen counterexamples, and use it to generate new counterexamples with a length between the shortest currently known counterexample and $f\cdot$ the longest known counterexample. The parameter f is tuneable and determines the maximum length of the generated sequences. The length of each new sequence is randomly chosen between the two bounds. Up to n new sequences are generated and used for checking.

Example

As an example, let's walk through what our method would do in the simple case of having one sequence:

```
('7', '7', '7', '3', '1', '7')
```

To keep things simple, we will take n-grams of size 1 and compute the transition matrix and start probabilities:

	1	3	7
1	0	0	1
3	1	0	0
7	0	0.33	0.67

Table 4.1: Markov transition matrix

	Start probability
1	0
3	0
7	1

Table 4.2: Start probabilities

Then, to generate a new sequence, our method will pick the first element according to the start probabilities: In this case it will always start with 7, since that is what our only training sequence starts with. Then, it will look in the transition matrix and move to either a 3 or a 7 with the probabilities 0.33 and 0.67 respectively. It will keep emitting new sequence elements in this way until a sequence of the desired length is reached.

Algorithm

input : hypothesis H, SUL S, input alphabet Σ, currently found counterexamples C, n-gram size s_n, query limit n_q, length increase factor f
output: Counterexample w if one exists, $None$ otherwise

shortest = length of shortest counterexample
longest = length of longest counterexample

M = fitMarkovChain(C, s_n)

for $i \in 0..n_q$ **do**
  newLength = $f \cdot$ random(shortest, longest)
  sequence = M.generate(newLength)

  $SULOutput \leftarrow \lambda_{\text{sul}}(sequence)$
  $HypOutput \leftarrow \lambda_{\text{hyp}}(sequence)$

  if $SULOutput \neq HypOutput$ **then**
    return *sequence*
  end
end

return $None$

Algorithm 8: Markov chain based equivalence checker

4.1.2. Experimental setup and results

To test the impact of using Markov chain based equivalence checking, we run experiments on RERS 2019 training problems 11, 12 and 13. We use TTT as learning algorithm, and use a stacked equivalence checker which first tries the Markov chain method with n-gram size 3, and then falls back to our smart W-method with a horizon of 6, 9 and 12 and most interesting first access sequence ordering. For comparison, smart W-method runs with horizon 6, 9, and 12 are also included. The end of each run is marked with the corresponding symbol seen in the legend.

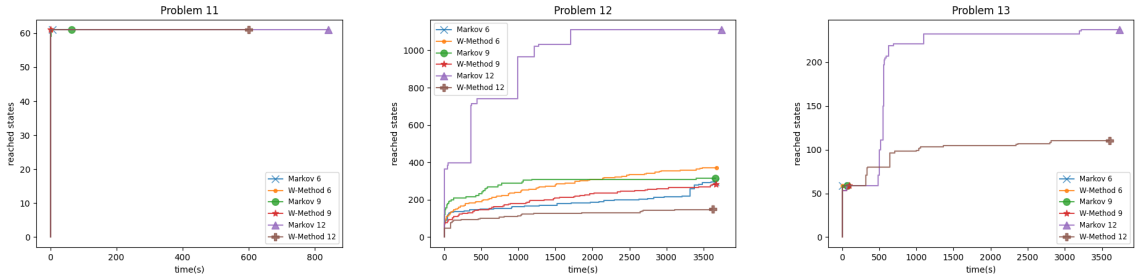

Figure 4.2: States found on the training problems of RERS 2019 using Markov equivalence checker

Problem 11, being the simplest problem, is solved quickly by all methods. Problem 12 sees a big jump in discovered states by the Markov run with depth 12. The runs with lower depths are not as effective, because problem 12 requires a rather large depth to find certain states in the first place. In problem 13, we see that all runs with a depth lower than 12 stop early on because they have exhausted the states they can find. The runs with depth 12 are more successful, and we can see that the Markov chain based method discovers states more quickly than the W-method based run.

To further verify the effects of the Markov chain based equivalence checker, we run it in a similar configuration using depths 6 and 12 on the reachability problems of RERS 2019. These are different from the ones

mentioned above, as those are the training problems of 2019. The results can be seen in Table 4.3.

	Method	States	Errors	Time (s)	Test queries
Problem 11	W 6	**41**	**20/20**	**1.189**	**79689**
	W 12	**41**	**20/20**	75.257	2545626
	M 6	**41**	**20/20**	2.310	154598
	M 12	**41**	**20/20**	42.549	2510334
Problem 12	W 6	55	15/27	**4.162**	**252738**
	W 12	221	16/27	timeout	76765626
	M 6	55	15/27	9.695	272564
	M 12	**1007**	**19/27**	timeout	3944243
Problem 13	W 6	361	17/19	**3314.201**	48330957
	W 12	155	**18/19**	timeout	101249245
	M 6	**404**	**18/19**	timeout	285502
	M 12	203	**18/19**	timeout	**130307**
Problem 14	W 6	**190**	**24/24**	**558.96** 0	23113867
	W 12	**190**	**24/24**	timeout	108040125
	M 6	177	**24/24**	timeout	324803
	M 12	172	**24/24**	timeout	**256296**
Problem 15	W 6	**254**	**43/55**	**1921.574**	71897209
	W 12	93	11/55	timeout	101424468
	M 6	165	38/55	timeout	229050
	M 12	129	25/55	timeout	**139317**
Problem 16	W 6	**212**	**12/21**	**731.852**	24289342
	W 12	**212**	**12/21**	timeout	95184726
	M 6	164	9/21	timeout	**168072**
	M 12	176	8/21	timeout	43079850
Problem 17	W 6	**716**	**41/41**	**3281.811**	182479968
	W 12	395	35/41	timeout	110915046
	M 6	299	33/41	timeout	**173232**
	M 12	292	36/41	timeout	219539
Problem 18	W 6	**671**	**0/32**	**3275.669**	145047075
	W 12	273	**0/32**	timeout	102617008
	M 6	273	**0/32**	timeout	467794
	M 12	291	**0/32**	timeout	**462160**
Problem 19	W 6	**864**	**22/36**	**3269.711**	122842708
	W 12	460	2/36	timeout	72115477
	M 6	219	1/36	timeout	205288
	M 12	247	12/36	timeout	**188159**

Table 4.3: Reachability 2019 Markov comparison table

These results indicate that the Markov chain equivalence checking method is effective for the first three problems. However, performance seems to suffer for the other problems, at least within the given time limit of one hour. It is hard to say if the Markov method would catch up with or overtake the W-method if given more time. But this could be an interesting avenue of further research for someone with more compute available.

4.2. Mutating counterexamples

Our second attempt at exploiting the structure found in these counterexamples is inspired by genetic algorithms. Genetic algorithms keep a population of individuals which all have a certain fitness score, and have the goal of maximizing this fitness. This is done by combining and mutating the individuals with a certain probability based on their fitness score, and keeping the ones with higher fitness.

In our case, the counterexamples would be the individuals in the population, but we have no way of assigning them a fitness value since they are all valid counterexamples and there is no signal that tells us how "good" or "bad" they are. However, we can still borrow the operators used by genetic algorithms to combine and mutate our population to find new counterexamples that look somewhat like the ones we already have. These new individuals will not all be valid counterexamples, but we can add the ones that are to our population and throw the ones out that are not.

4.2.1. Intuition

In this chapter, our goal is to take a set of counterexamples, find a pattern in them, and using that pattern to generate new, unseen counterexamples. As said before, extracting a pattern from only positive examples is challenging, so we sidestep the need to do this by taking the counterexamples we have and mutating them using genetic operators to create new test cases to try. This does not directly learn a pattern, but we assume that since the newly generated sequences are made by taking parts of the already existing counterexamples and assembling them together along with some duplications and deletions, they will follow the already established patterns quite well.

One thing that caught our attention while looking at the counterexamples found by a normal equivalence check, is that there did not appear to be one single pattern that could describe all of them. For example, process of learning the state machine shown in Fig. 4.1 generated the counterexamples shown earlier, but also the following ones:

```
('4', '1')
('4', '1', '9', '6')
('4', '4', '8', '3')
('4', '6', '5', '3', '2')
('4', '6', '5', '3', '3', '2')
('4', '4', '1', '5')
('5', '5')
('5', '5', '7')
('5', '6', '10', '1', '5')
('5', '5', '6', '5', '7', '7', '1', '5')
```

While this sample by itself may not be enough to see a clear pattern, these counterexamples are clearly not following the same pattern as the ones mentioned at the start of this chapter. Therefore, it may be helpful to not put all counterexamples in the same population where they will be mutated and combined with each other, but to cluster them and keep them separate to ensure the patterns do not get mixed up needlessly. We will investigate whether or not this pre-clustering step is helpful.

Also note that counterexamples might not be minimal, e.g. they can contain redundant steps that could be omitted without changing the output, therefore we assume TTT will benefit the most from this method since it is designed to handle non-minimal counterexamples quite well.

4.2.2. Method

Mutators

In our method for mutating counterexamples, we use two operators. For mutation, we use duplication and deletion, which picks a random point in a counterexample and duplicates it with a certain probability, or removes it from the individual. We found that biasing this probability towards duplicating rather than deleting is helpful since it will try growing longer counterexamples from the ones that are already present, and intuitively the shorter counterexamples are found first.

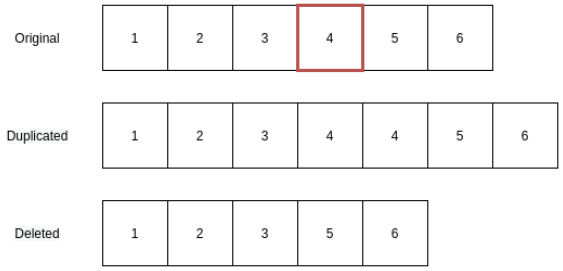

Figure 4.3: Mutation example. The square marked red is the element being mutated

Our other operator is single point crossover, where on each of the parents a random point is picked, and a new individual is assembled from the part left to the chosen point of one parent, and the part right to the point on the other parent. Usually, the single point crossover operator is implemented by picking a single point that is the same on both parents to keep the length of the children the same. However, since we are not working with a fixed size for the individuals, we modify it slightly by picking a different point on both parents to allow for longer and shorter offspring.

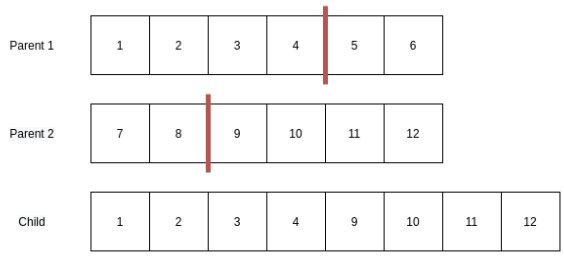

Figure 4.4: Crossover example. The red lines indicate crossover points

Equivalence checker

To be able to mutate counterexamples, we need to find some counterexamples to mutate in the first place. Since the mutation based equivalence checker is not very effective at finding the initial few counterexamples, it is necessary to stack it with another method like done in [41]. We use a stacked equivalence checker with the mutation based checker first, and a smart W-method second. With reasonable parameters, the mutation based checker finishes quickly, so it does not cause significant overhead by being put first.

The full mutating equivalence checker algorithm can be seen in 9.

input : hypothesis H, SUL S, a list of counterexamples C, crossover probability P_c, mutation probability P_m, maximum population size n
output: Counterexample w if one exists, $None$ otherwise

```
/* Mutate individuals until population size n */
new_population = ∅
while |new_population| < n do
    if random() ≤ P_c then
        // randomly pick two parents
        parent1, parent2 ∈ C
        individual = crossover(parent1, parent2)
    else
        // or randomly pick an individual
        individual ∈ C
    end
    if random() ≤ P_m then
        individual = mutate(individual)
    end
    new_population = new_population ∪ {individual}
end
/* Finally, use the newly generated traces in an equivalence check */
for trace ∈ new_population do
    SULOutput ← λ_S(trace)
    HypOutput ← λ_H(trace)
    if SULOutput ≠ HypOutput then
        return w
    end
end
return None
```

Algorithm 9: Mutating equivalence checker

4.2.3. Experimental setup and results

We again learn state machines of the RERS 2019 reachability training problems 11, 12 and 13. We use TTT with the mutating equivalence checker in 9 stacked with the smart W-method with horizon 6, 9 and 12. For comparison we also include just the W-method by itself in the same configuration.

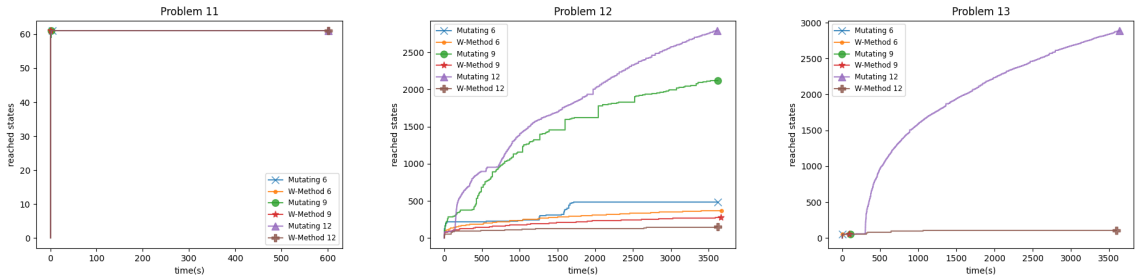

Figure 4.5: States found on the training problems of RERS 2019 using mutating equivalence checker

Like before, problem 11 is solved quickly. Problem 12 and 13 both see a very significant speedup when the depth is properly set. All in all, after around one hour the mutating method finds over 2500 states in problem 11 and almost 3000 in problem 13, compared to just over 1000 and almost 250 found by the Markov based method. We can conclude the mutating equivalence checker is superior on these problems.

To more thoroughly check the performance of the mutating equivalence checker, we also ran a comparison on the RERS Reachability 2019 problems. To clarify, these are not the same problems as the training

problems used in Figure 4.5. The columns labelled "W n" are W-method with depth n. The columns labelled "M n" are the mutating equivalence checker stacked with a W-method of depth n. The max values are highlighted in each row. For each run, the learning process was terminated after one hour if it did not finish by then.

	Method	States	Errors	Time (s)	Test queries
Problem 11	W 6	**41**	**20/20**	**1.189**	**79689**
	W 12	**41**	**20/20**	75.257	2545626
	M 6	**41**	**20/20**	2.504	99190
	M 12	**41**	**20/20**	84.156	2813390
Problem 12	W 6	55	15/27	**4.162**	252738
	W 12	221	16/27	timeout	76765626
	M 6	55	15/27	5.086	**235406**
	M 12	**2999**	**19/27**	timeout	3094162
Problem 13	W 6	361	17/19	**3314.201**	48330957
	W 12	155	**18/19**	timeout	101249245
	M 6	1167	**18/19**	timeout	**4111261**
	M 12	**1260**	**18/19**	timeout	16143889
Problem 14	W 6	**190**	**24/24**	**558.960**	**23113867**
	W 12	**190**	**24/24**	timeout	108040125
	M 6	**190**	**24/24**	585.915	25157623
	M 12	**190**	**24/24**	timeout	389705621
Problem 15	W 6	**254**	**43/55**	**1921.574**	**71897209**
	W 12	93	11/55	timeout	101424468
	M 6	**254**	**43/55**	2023.106	76386831
	M 12	192	42/55	timeout	355449398
Problem 16	W 6	**212**	**12/21**	**731.852**	**24289342**
	W 12	**212**	**12/21**	timeout	95184726
	M 6	**212**	**12/21**	735.475	24630033
	M 12	**212**	**12/21**	timeout	310215182
Problem 17	W 6	**716**	**41/41**	**3281.811**	182479968
	W 12	395	35/41	timeout	**110915046**
	M 6	**716**	**41/41**	timeout	223956043
	M 12	644	**41/41**	timeout	373634871
Problem 18	W 6	671	0/32	**3275.669**	145047075
	W 12	273	0/32	timeout	**102617008**
	M 6	764	11/32	timeout	160871307
	M 12	**778**	**17/32**	timeout	298504299
Problem 19	W 6	864	**22/36**	**3269.711**	122842708
	W 12	460	2/36	timeout	**72115477**
	M 6	**868**	**22/36**	3545.410	126318073
	M 12	810	18/36	timeout	270496480

Table 4.4: Reachability 2019 mutating comparison table

We can see in Table 4.4 that the mutating equivalence checker performs on par or better than the W-method equivalence checker by itself. The benefit gained over the standard W-method does seem to be dependent on the problem. Interestingly, problem 12 and 13 show a large increase in the amount of states found with the mutating equivalence checker, while the rest is mostly just on par or marginally better.

To investigate what was causing this disparity, we visualized the learned state machines of problem 12, 13, 18 and 19 in Appendix A.1. To help visibility, we limited the plotted states by access sequence length per problem, and left out "invalid input" and "error" self loops. Evidently, the structure of problem 18 and 19 is less suited for the mutating equivalence checker since they contain a lot of "traps" or branches that lead to a state where all inputs are invalid. These catch the sequences generated by the mutating equivalence checker, causing no further states to be explored. This is as opposed to problem 12 and 13, which contain far fewer of these structures to get stuck in.

4.2.4. Can pre-clustering help?

We also check if pre-clustering the counterexamples can make this method even more effective. Intuitively, since mixing up the patterns will degrade performance, this should help. Whether or not this is the case in practice will be investigated in this section.

Clustering

To divide our counterexamples into populations that hopefully consist of individuals generated by the same or at least similar patterns, we calculate pairwise distances between the counterexamples using a longest common subsequence distance metric, and cluster them using HDBSCAN [29] using a precomputed distance matrix. This method was chosen because HDBSCAN does not require the user to specify a predetermined amount of clusters, and instead only requires setting a parameter indicating the minimum cluster size that we care about. However, since our distance metric does not necessarily capture the similarity between our counterexamples very well, we first project our counterexamples to a lower dimensional space using uniform manifold approximation [30]. UMAP makes the assumption that the data it is fed lies on a uniformly distributed riemannian manifold, and that the manifold is locally connected. Since our chosen distance metric does accurately capture the similarity of two counterexamples when there are only a few changes between them, this locality allows UMAP to project the data into clusters in a way that makes sense intuitively. Another benefit of this we observed is that HDBSCAN has a much easier time picking out the clusters since the UMAP projection prioritizes the local structures in our data and smooths out the global relationships making the clusters more coherent.

An example of a 2d representation of counterexamples found during a run on Problem 12 from the sequential training problems of RERS 2019 can be seen in Figure 4.6

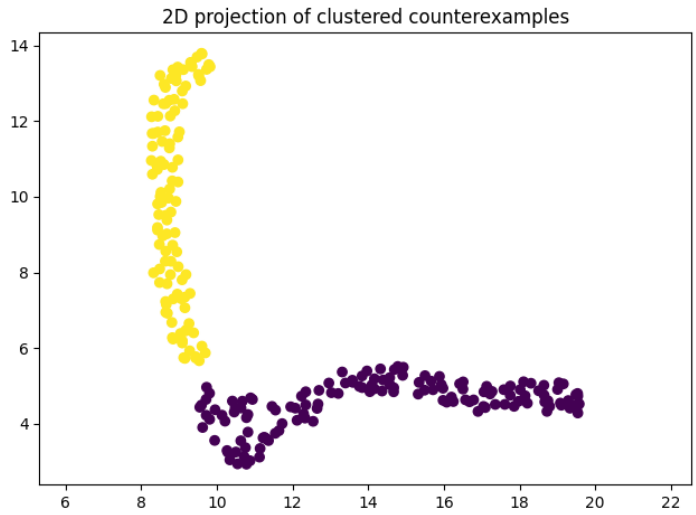

Figure 4.6: UMAP projected and clustered counterexamples

We can check if the projection and the cluster assignment make sense. This projection tells us there are two major patterns in the set of counterexamples we have, hence the two long stretched clusters. If we check what counterexamples got assigned to the yellow cluster, we see the pattern with $(7,7...7,7,3,1,7,7...7,7,1,5...)$ is present in almost all points assigned to the cluster. Shorter sequences at the bottom, with the longest at the top. The long cluster stretching to the right consists of the pattern $(4,4,8,9,4,8,9,1,9,3,1,7,7,7...7,7,3,1,7,7,7...)$. The smaller, round-ish cluster at the bottom left consists of shorter counterexamples that are most likely not long enough to accurately determine a pattern of. Ideally this would be a separate cluster, which could most likely be archieved with some tuning of the HDBSCAN parameters.

An interesting thing to note is that the two clusters correspond to a pattern seen in the state machine learned for Problem 12 (see Figure 4.7). If we draw the graph representing the hypothesis learned after a while, we see two "branches" of a repeating pattern, which correspond to the two stretched clusters in the umap projection.

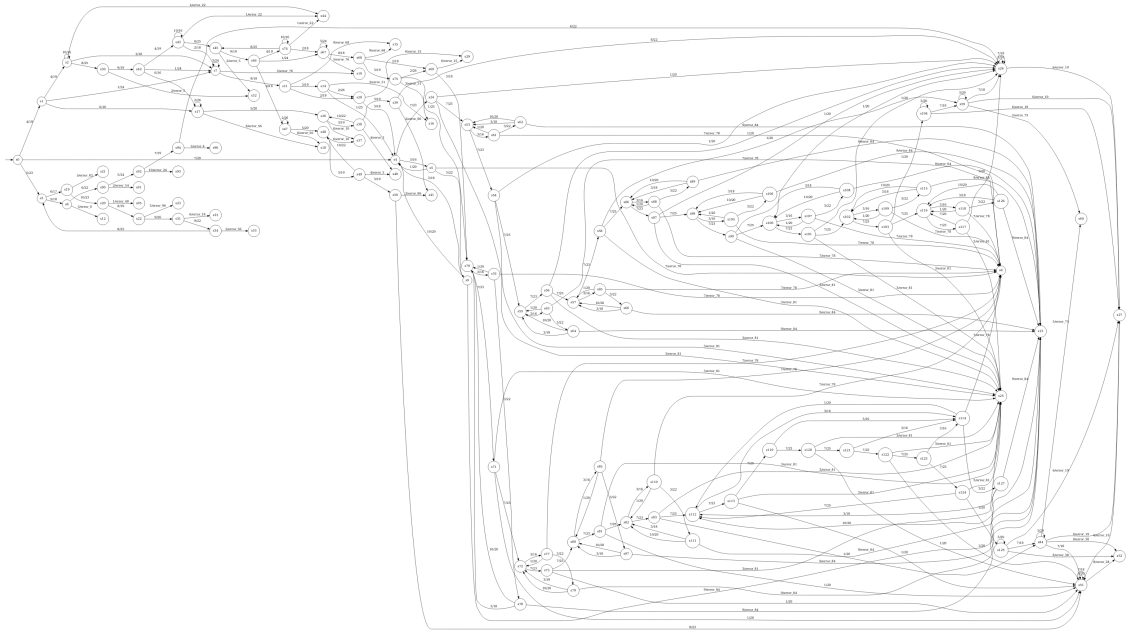

Figure 4.7: Problem 12 intermediate hypothesis. Note the two repeating patterns on the right.

input : hypothesis H, SUL S, a list of counterexamples C, crossover probability P_c, mutation probability P_m, maximum population size n
output: Counterexample w if one exists, $None$ otherwise

```
/* First we project and cluster the provided counterexamples */
distmat = calculate_distance_matrix(C)
projected = UMAP(distmat)
clusters = HDBSCAN(projected)
/* Then, for each cluster, mutate individuals until population size n */
populations = []
for cluster ∈ clusters do
    new_population = ∅
    while |new_population| < n do
        if random() ≤ P_c then
            // randomly pick two parents from cluster
            parent1, parent2 ∈ cluster
            individual = crossover(parent1, parent2)
        else
            // or randomly pick an individual
            individual ∈ cluster
        end
        if random() ≤ P_m then
            individual = mutate(individual)
        end
        new_population = new_population ∪ {individual}
    end
    populations.append(new_population)
end
/* Finally, use the newly generated traces in an equivalence check */
for population ∈ populations do
    for trace ∈ population do
        SULOutput ← λ_S(trace)
        HypOutput ← λ_H(trace)
        if SULOutput ≠ HypOutput then
            return w
        end
    end
end
return None
```

Algorithm 10: Mutating equivalence checker with clustering

Comparison vs no clustering

While this method still provides a significant speedup compared to using a plain W-method equivalence checker, there are certain downsides to it. Once the collection of counterexamples grows, the clustering starts to take more and more time since calculating the distance matrix is an $O(n^2)$ operation, and calculating the longest common subsequence between two sequences is $O(nm)$, where n and m are the lengths of the respective sequences. This makes the clustering method unsuitable for larger problems, but fortunately the method is still effective with the clustering turned off. A comparison vs no clustering can be seen in Figure 4.8

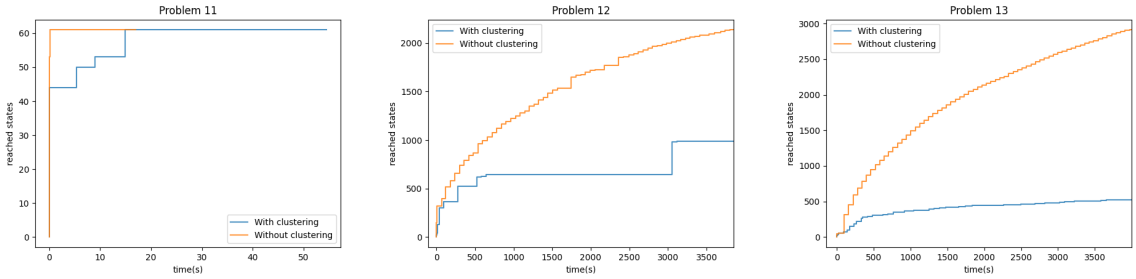

Figure 4.8: Clustering vs no clustering on RERS 2019 using mutating equivalence checker

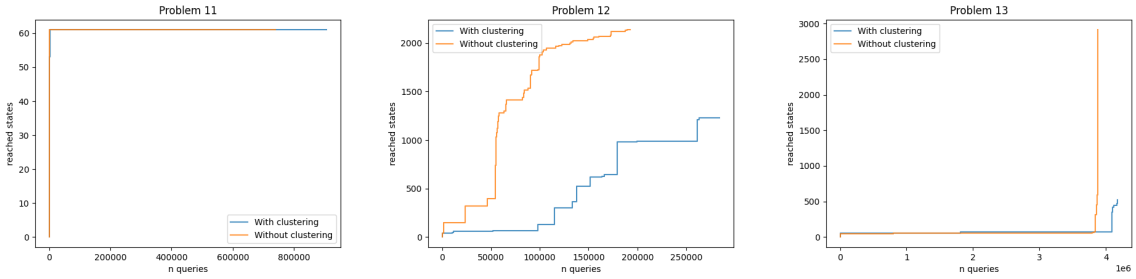

Figure 4.9: Amount of queries vs States found

We can clearly see that, going purely by wall-clock time, pre-clustering does not lead to better performance on these problems. To see what causes this drop in performance, we plotted the amount of queries done vs. the amount of states found in Figure 4.9. If the overhead of the clustering step was causing the drop in performance, we would see at least a roughly equal amount of states found per query. That is not the case, and thus the clustering step not only degrades performance time-wise but also makes the algorithm less effective at finding new states.

5

Using white/grey box testing techniques

White and grey box testing techniques usually involve some kind of source code analysis or instrumentation. Since the RERS challenge gives us access to the source code of the problems, we investigate techniques to take advantage of this and improve upon the standard black box active state machine learning algorithms. Previous work [24] [36] has investigated using fuzzers in combination with active learning algorithms to search for interesting test cases to use during equivalence checking. The results of this work are promising so we investigate these techniques and integrate them into our state machine learning library. Another technique that could be applicable in this context is symbolic execution. However, after some initial attempts of using KLEE [7] and angr [34] on the RERS problems this was deemed to computationally expensive and it was decided to focus on fuzzing instead.

In this chapter, we will first look into two popular fuzzers, AFL and LibFuzzer. We will explain how we set them up to use on the RERS problems, and investigate their performance on the reachability problems we have been using so far. After that, we will explain how fuzzers can be used to augment the process of learning state machines, or more precisely, how the corpus of interesting inputs discovered by a fuzzer can help the equivalence checking step in the learning process.

5.1. Fuzzing

For those unfamiliar, fuzzing is a technique used in software testing that tries to find flaws in a given piece of software by providing it with input data created by a fuzzer. In it's simplest form, this could be done by randomly flipping bytes in a known valid input until something interesting (e.g. a crash) happens. In practice, most fuzzers are a lot smarter about it and use instrumentation to gain knowledge of the coverage reached by the generated inputs.

AFL

A well known example of this is AFL[1], a grey-box fuzzer that uses instrumentation and a genetic algorithm to create as many interesting different inputs as possible. AFL has been used to find bugs in many different well known software projects, like Xorg server, OpenSSL, and SQLite. In the work by Janssen [24] AFL was used to find interesting test cases to use for equivalence checking, which greatly sped up the learning process.

One downside of AFL is that it essentially runs two separate processes: the main AFL process that runs the fuzzer, and another process that is the application being fuzzed. This means that AFL has to restart the second process for every test case it runs, which is a relatively expensive operation if you want to absolutely go as fast as possible. A lot of work has been done to negate the impact of this, for example AFL employs what is called a forkserver, which loads the program being fuzzed and forks it from a clean state instead of building it up from scratch for every run. Furthermore, it is possible to use "persistent mode", which uses a special macro __AFL_LOOP to create a loop that processes multiple inputs without having to start or fork a new process.

Another downside of AFL's approach of using two separate processes is that the fuzz cases have to be sent to the process that is being fuzzed either through stdin/stdout or through a file. This is generally not a problem if you are running one or just a few fuzzers in parallel, but it prevents effective scaling on machines

[1] https://github.com/google/AFL

with many threads/cores since all the communication goes through the kernel. This means that, once you throw enough cores at the problem, you will not gain any additional benefit since most of your time is spent waiting in the kernel [2].

libFuzzer

Similar to how we improved our query speed on the RERS challenge problems in section 3.2.1 by switching to in-process querying, fuzzing can be done more efficiently using in-process fuzzing. A fuzzer capable of doing in-process fuzzing is libFuzzer [33], which works by linking with the target process directly and thus avoids both the overhead of restarting, and inter-process communication entirely. To use libFuzzer, one simply needs to include a function "LLVMFuzzerTestOneInput" into the target program, which runs a single testcase, and compile it using LLVM with the flag -fsanitize=fuzzer. This results in a binary which, when run, fuzzes the process until a crash is found, which is then reported.

5.1.1. AFL vs libFuzzer on RERS

To compare AFL and libFuzzer, we will evaluate their performance on the reachability training problems of the RERS 2019 challenge. We describe how we modified the RERS code to gain the maximum possible performance out of both fuzzers, and compare the error states found over time by both methods.

AFL RERS modifications

To get the best possible performance out of AFL, we modify the code of the RERS problems to leverage persistent mode by adding a reset function as before, and creating a loop to process the test cases. The modification to the main loop can be seen in Listing 5.1. Instead of terminating the program on an out-of-alphabet input, we break from the loop which processes the input, reset, and proceed to processing the next input sequence. We also extract the input alphabet of the problem and use that to generate a dictionary file which is passed to AFL.

```
int main()
{
    char buf[1024*1024];

    while(__AFL_LOOP(10000000))
    {
        reset();

        // Read from stdin (fd 0)
        size_t n = read(0, buf, 1024*1024);

        // operate eca engine
        for (size_t i = 0; i < n; i++) {
            int input = buf[i];

      if((input != 2) && (input != 10) && (input != 9) && (input != 5) && (input != 3) && (input != 4) &&
     (input != 1) && (input != 7) && (input != 8) && (input != 6))
                break;

            calculate_output(input);
        }
    }
}
```

Listing 5.1: AFL modified RERS main loop

A nice property of AFL is that it saves any crashes it encounters in a separate folder, and keeps running indefinitely. This means we can keep the standard error checking code which calls assert(false) once an error state is reached and terminates.

libFuzzer RERS modifications

As mentioned before, to be able to use libFuzzer on the RERS problems we have to write a function called "LLVMFuzzerTestOneInput". This is a relatively simple modification only requiring us to define how a single input should be handled by the program. We simply replace the main function with the code seen in listing 5.2. Same as before, we also extract the alphabet out of the source code and create a dictionary for libFuzzer to use.

[2] https://gamozolabs.github.io/fuzzing/2018/09/16/scaling_afl.html

```
int LLVMFuzzerTestOneInput(const char *Data, size_t Size) {
    reset();

    // operate eca engine
    for (size_t i = 0; i < Size; i++) {
        int input = Data[i];

      if((input != 1) && (input != 2) && (input != 3) && (input != 4) && (input != 5) && (input != 6))
          return -2;

        calculate_output(input);
    }
    return 0;
}
```

Listing 5.2: libFuzzer function

One downside of libFuzzer is the fact that it terminates after a single crash (or a specified amount of crashes) is found , and to the best of our knowledge there is no easy way to run it indefinitely like AFL. Therefore we also have to modify the error checking code to not abort on reaching an error state. Instead it should simply reset the program and continue to the next input. The consequence of this is that we do not get a separate folder containing crashing inputs, but all interesting (coverage increasing) inputs get put in a single corpus directory. LibFuzzer also overwrites test cases once it finds a smaller input with the same coverage. This would be a nice feature for normal fuzzing jobs, but we want to be able to see the exact time each error state has been reached for performance evaluation reasons. This required us to write a separate file watcher process to copy the error-reaching test cases out of the corpus directory, saving their "last modified" date.

Performance comparison

To compare the performance of both fuzzers on our problems, we need to go with a meaningful measurement of performance. For both the LTL and reachability RERS problems we are interested in getting as much coverage throughout the problems as quickly as possible, but directly measuring for example block or branch coverage may not be representative of the tasks we are trying to solve. A better solution would be to learn a state machine using the traces generated by the fuzzer for each timestep in the evaluation, but this is prohibitively expensive. In the end we decided to use the amount of reached errors over time as a metric, since this is easy and computationally inexpensive to measure, and directly representative of our performance on the reachability track. We are making the assumption here that performance on the LTL problems will not differ much, or at least be in the same ballpark.

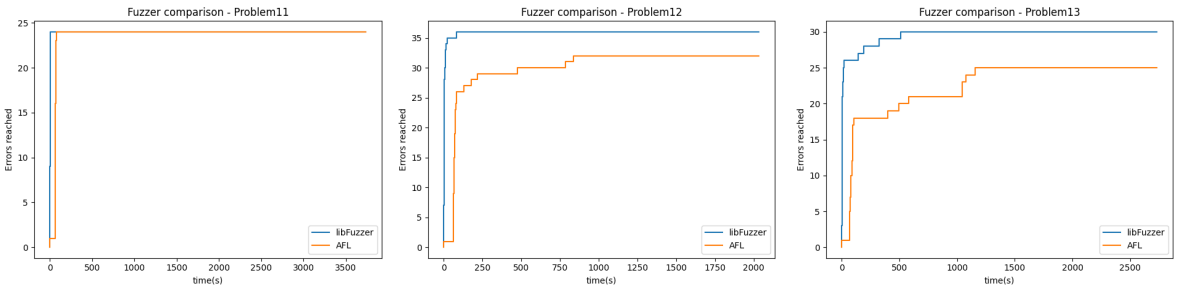

Figure 5.1: Errors reached over time on the training problems of RERS 2019

As we can see in Figure 5.1, the libFuzzer fuzzer outperforms AFL on all but the simplest problems. All error states in problem 11 are reached by both fuzzers within seconds. However, AFL struggles to find all error states within an hour on problem 12 and 13, while libFuzzer finds all reachable error states within minutes. Whether this is purely due to the speed increase libFuzzer gives us or because of more effective instrumentation or something else is something open to further research.

We also checked if our intuition about libFuzzer being the better choice simply because it was faster was true. We measured the executions per second over time for both fuzzers for all problems, and plotted them in Figure 5.2 and 5.3. AFL conveniently logs data during fuzzing which we could plot directly, while we had to be a little creative with libFuzzer since it does not include timestamps in it's log file. We fixed this by piping

the log output of libFuzzer through ts[3], which prepends a timestamp to each line written. To track the total number of executions, we integrated the executions per second graphs using scipy's traps function.

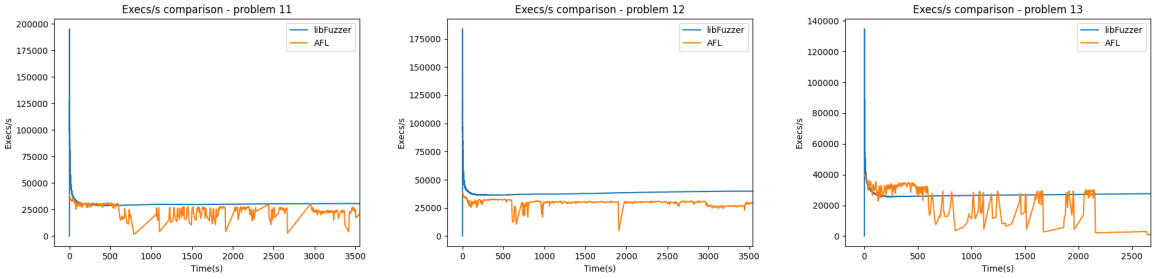

Figure 5.2: Executions per second on the training problems of RERS 2019

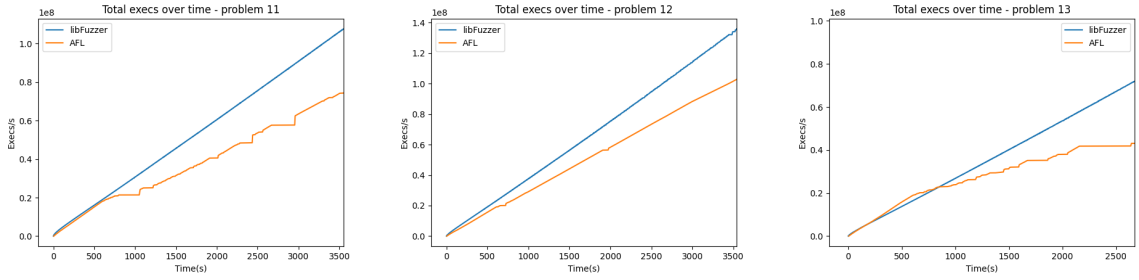

Figure 5.3: Total fuzz cases ran on the training problems of RERS 2019

One thing that immediately drew our attention was the consistent performance of libFuzzer. It is likely that it keeps a rolling average internally explaining the smoothness, but during the fuzzing AFL's execs/s value could be seen dropping low quite often, supporting what is seen in the graphs. All in all, the performance difference is smaller than we expected at first. This can most likely be explained by the fact that only 3 AFL processes were running in parallel on one machine, thus preventing the largest bottleneck of kernel IO. Even then, libFuzzer is still faster in all cases except at the start of fuzzing problem 13, and it also finds the reachable error states more quickly. It is interesting that is not likely to be explained purely by the difference in exectution speed, so libFuzzer's instrumentation or mutation engine is probably also more suitable for the type of problem we are running it on.

5.2. Using fuzzing results for equivalence checking

Next to verifying the crashing testcases and seeing what errors they reach, running a fuzzer standalone is all that is needed to do pretty well in the reachability challenges of RERS 2019. However, for the LTL problems we require state machines to later run a model checker on. Therefore we use the corpus of testcases found by the fuzzer as traces to ask during equivalence checking. This process is about as simple as can be, but for completeness is still included in Algorithm 11.

[3]https://joeyh.name/code/moreutils/

input : hypothesis H, SUL S, input alphabet Σ, corpus (set of input traces found by fuzzing) T
output: Counterexample w if one exists, $None$ otherwise

for $testcase \in T$ **do**
  `// Strip out non-alphabet characters from the testcase`
  trace = $[x | x \in testcase$, if $x \in \Sigma]$
  $SULOutput \leftarrow \lambda_{\text{sul}}(trace)$
  $HypOutput \leftarrow \lambda_{\text{hyp}}(trace)$
  if $SULOutput \neq HypOutput$ **then**
    return $trace$
  end
end
return $None$

Algorithm 11: Corpus equivalence checker

Results

To see how well the equivalence checking using a fuzzer corpus works, we evaluated it on the 3 reachability training problems of RERS 2019 again. We use libFuzzer, and for each problem the fuzzer was ran for 15 minutes. Data from a run using the W-method with depth 8 (11 for problem 13) is also plotted for comparison.

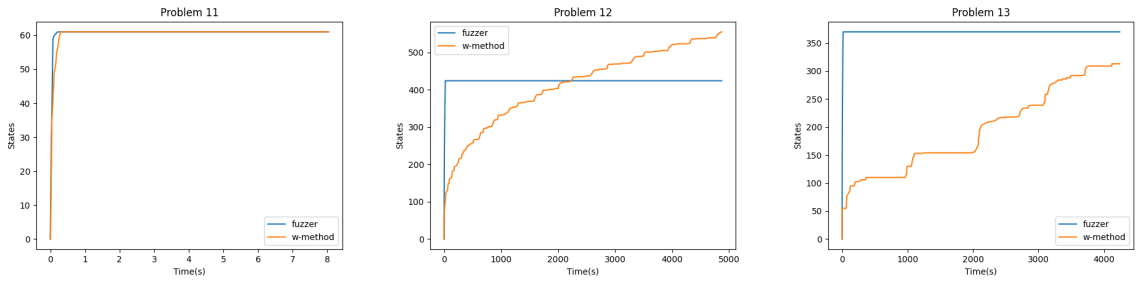

Figure 5.4: States found on the training problems of RERS 2019

As before, problem 11 is the easiest and finishes learning completely in under a second in both cases. Problem 12 is more interesting to look at: while the fuzzer was able to find all error states, the traces provided by the corpus alone are not enough to learn a model of the complete behavior of the program. This is confirmed by the W-method based learner overtaking the fuzzer based learner after running long enough. This means that we will need to take additional learning steps besides fuzzing in case we want a model that captures the complete behavior of the RERS programs. Problem 13 is a similar story, only since it is even bigger than problem 12 and requires a higher W-method depth to find more than a handful of states, it was not able to find more states than the fuzzer based method within the given time.

5.3. Is fuzzing enough?

The results in Figure 5.4, particularly Problem 12, makes one wonder if the traces found by fuzzing alone are enough to capture the complete behavior of the program. Seeing as the W-method based approach overtakes the amount of states found by the fuzzer based approach after some time, we have to assume the model learned with just the fuzz cases as equivalence check is incomplete. However, we also know that libFuzzer manages to find all reachable error states in Problem 12 within minutes. To see why this is the case, we need to delve into how fuzzers measure coverage and determine what makes a trace "interesting" to a fuzzer or not.

5.3.1. Fuzzer instrumentation

Both AFL and libFuzzer determine whether inputs they try are interesting or not by looking at the coverage these inputs reach. They can determine the coverage reached by a certain input by leveraging the instrumentation they put in the programs they fuzz. On a high level, this instrumentation consists of a "guard" function

call on each edge between basic blocks in the control flow graph of the program that is being fuzzed. To determine the coverage of a certain input, the fuzzer counts how many times each guard is triggered. This is to make sure that loops can be counted as well.

```
int foo(int a) {
    if (a < 10) {
        a = 10;
    }
    return a;
}
```

Listing 5.3: Fuzzer example

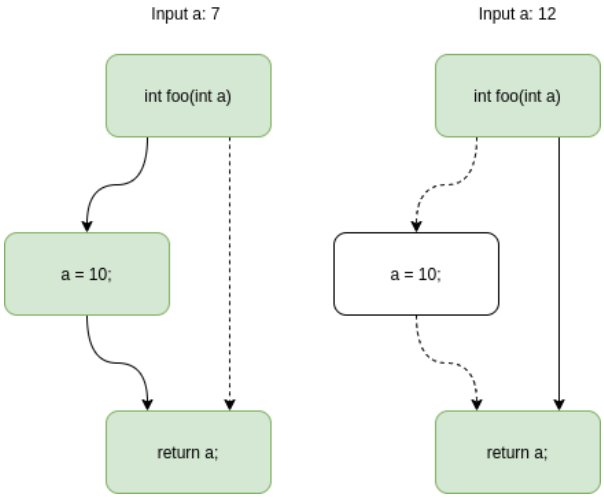

Figure 5.5: The two paths through the example code

Let's look at a simple example, adapted from here [4]. Let's say we have a function we want to fuzz (listing 5.3). We can see this simple example consists of three basic blocks. In Figure 5.5 we can see the two possible paths through the program, along with the basic blocks (green) and edges (solid) they hit.

We know that libFuzzer is successful in finding all the reachable error states in the RERS 2019 training problems. If we take a look at the code that leads up to these error states (Listing 3.2), we can see why the fuzzer is successful in finding them: each occurrence of the __VERIFIER_error function call would clearly get it's own guard assigned, and thus each time an error is triggered new coverage would be found and the input that triggered it would be added to the corpus.

For a more in depth explanation of fuzzer internals, please see the background section of the Angora paper [9].

5.3.2. Missed behavior

If we visualize the state machines learned on Problem 12 by the W-method and the fuzzing equivalence checker, we see that the W-method captures more behavior than the fuzzed traces alone. For clarity, we limited the states to the ones with an access sequence shorter than 10, and mark an interesting path (of repeating 7's as input) in green.

[4] https://farlopa.gratis/libfuzzer-internals-2/

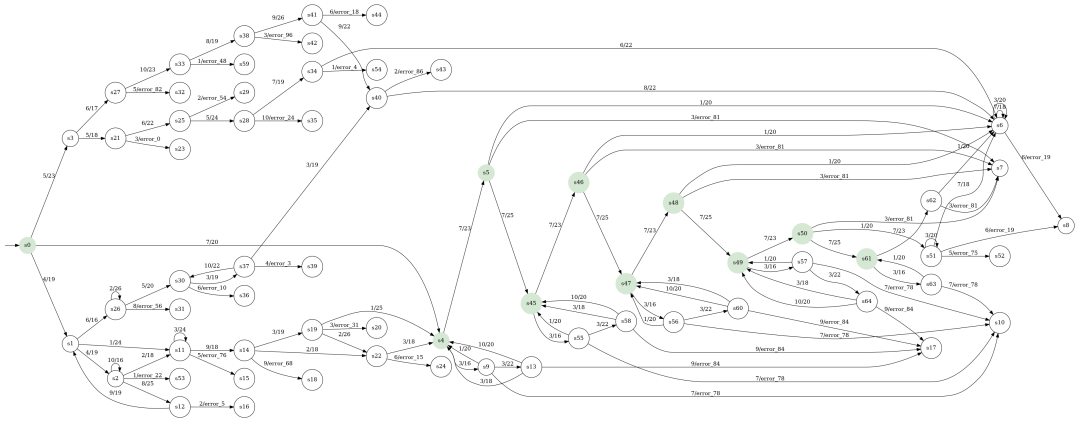

Figure 5.6: Problem 12 W-method

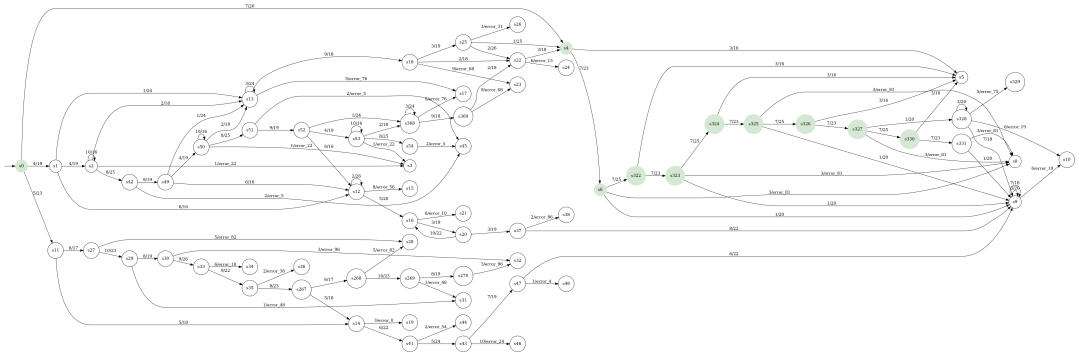

Figure 5.7: Problem 12 Fuzzer

We marked the same path in both figures, and by comparing we can easily see that Figure 5.6 shows more complex behaviour than Figure 5.7. In the fuzzer-based graph, the tail of the sequence of 7's is missing the looping structure seen in the W-method based one. This is most likely due to the instrumentation of the fuzzer not picking up on these structures, or it is somehow determined that these sequences are not "interesting" according to the metrics in the fuzzer.

So with this, we have shown that learning state machines using *only* a fuzzer-based equivalence checker is not enough to capture the complete behavior of this problem. In the context of the reachability task, this is not an issue since the fuzzer's instrumentation makes sure all reached error states are captured. In the context of LTL checking, these missing states have the potential to lead to incorrect answers, and thus we will need to augment the fuzzer based equivalence checker (with for example a fallback to the W-method) to make sure the learned model is correct.

6

Putting it all together for RERS

Over the course of the previous chapters we have investigated several ways of improving the learning time of active state machine learning on the 2019 RERS training problems. We concluded that for the reachability track, running a libFuzzer based fuzzer is the most successful approach since it is not a requirement to actually learn a state machine to see what errors are reachable. On top of that, if we relied on a purely active state machine learning based approach for the reachability track we would be missing out on useful information since we would be using only black box techniques. It is clear that the grey/white box approach of using a fuzzer has a significant advantage since it can make use of instrumentation instead of just relying on the output of the program.

The problems in the LTL track, which we only have briefly discussed up until now, definitely benefit from learning state machine models. As discussed in chapter 3, in the LTL track we are given obfuscated code similar to that from the reachability track and are required to prove or falsify a list of given LTL formulae. To do this, having a as-complete-as-possible overview of the behavior of the programs in the form of a state machine is invaluable.

In this chapter we will describe our approach to the reachability and LTL challenges of RERS 2020, and analyse our performance on the problems of RERS 2019.

6.1. LTL

As mentioned before, the LTL problems require us to prove or falsify certain LTL formulae on a given system. The RERS challenge does not set any rules or limits on how we can do this, so we chose an active learning based approach combined with a model checker capable of working with LTL formulae. After trying mcrl2 [18] and figuring out it does not support LTL checking, trying LTSMin [25] and getting confused by the documentation, we chose NuSMV [11] as model checker since it is relatively easy to use, fast, and directly supports LTL model checking out of the box.

The theory of LTL model checking is outside the scope of this thesis so we will not go into that. We will treat the model checker as a black box we can feed our hypothesis state machine and the LTL formulae into, after which it will tell us which of the LTL formulae it was able to falsify and find a counterexample to.

6.1.1. NuSMV conversion

NuSMV uses a specialized input language to describe synchronous or asynchronous finite state systems[1]. This means we need to somehow translate our Mealy machine hypotheses into the NuSMV input language. Fortunately, this is a relatively simple task. One thing to keep in mind is that the RERS challenge uses alternating LTL semantics. This means that in the input/output traces, the input and output steps are temporally separate, e.g. the output occurs after the corresponding input. If we were using synchronous LTL semantics, we would assume the input and corresponding output happen at the exact same point in time.

To deal with the alternating semantics, we need to add an extra state between each transition in the Mealy machine hypothesis, since the output essentially takes an extra step. This means that we are translating our Mealy machine into a labeled transition system, which is essentially a Mealy machine with only a single output or input label on the transitions. An example of such a conversion can be seen in Figure 6.1.

[1] http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf

Figure 6.1: Example translation from a Mealy machine (left) to labeled transition system (right)

Once we have converted our Mealy machine into a labeled transition system, we still need to translate it to the NuSMV input language. This is done by a python script available in our git repository [2]. The result of translating the example Mealy machine in figure 6.1 can be seen in Listing 6.1.

```
1  MODULE main
2  VAR
3          state : {q0,q0_o_a,q1,q1_o_a};
4          output: {A,a};
5  ASSIGN
6          init(state) := q0;
7          next(state) := case
8                          state = q0  & output = a: q0_o_a;
9                          state = q0_o_a : q1;
10                         TRUE : state;
11         esac;
12         init(output) := {a};
13         next(output) := case
14                         state = q0 & next(state) = q0_o_a : A;
15                         TRUE : output;
16         esac;
```

Listing 6.1: Example NuSMV code

The NuSMV input language requires us to define the variables we are interested in and the values they can assume, as well as the transitions they can make and on what conditions those transitions apply. We define the next state based on the current state and current "output" (which can be either an input or output in the original Mealy machine); if the current state is q0 and the current output (which was an input in the original Mealy machine) is 'a', we know we need to go to the additional output state that comes after q0, named q0_o_a by our script. The next output is defined in a similar way. The last line containing TRUE is a catch-all fallthrough option keeping the state and output the same. This is required by NuSMV.

Translating the LTL formulae

One final thing we need to do before we can let NuSMV check the LTL properties on our Mealy machines, is translate the LTL formulae into a format NuSMV understands. RERS specifies their temporal operators in a slightly different way than NuSMV, but luckily the translation is quite straightforward and can be seen in 6.1.

Name	RERS	NuSMV	Description
Next	X ϕ	X ϕ	ϕ has to hold after the next step
Eventually	F ϕ	F ϕ	ϕ has to hold at some point in the future (or now)
Globally	G ϕ	G ϕ	ϕ has to hold always (including now)
Until	ϕ U ψ	ϕ U ψ	ϕ has to hold until ψ holds (which eventually occurs)
Weak until	ϕ WU (or just W) ψ	ψ R (ϕ | ψ)	ϕ has to hold until ψ holds (which does not necessarily occur)
Release	ϕ R ψ	ϕ V ψ	ϕ has to hold until ψ held in the previous step

Table 6.1: RERS LTL operators and their NuSMV equivalents

Besides the translations listed in Table 6.1, some other steps need to be taken:

- Replace "true" and "false" with "TRUE" and "FALSE"

[2] https://github.com/TCatshoek/lstar/blob/master/util/mealy2nusmv.py

- Replace the input and output variables (iA, iB, iC, oU, oV, oW, etc.) with (output = $variable identifier). The variable identifier can be found in the ProblemX_alphabet_mapping.txt file provided by RERS.

With all these steps done, we can write the LTL constraints in our NuSMV file in a line that looks like the following:

```
LTLSPEC NAME rule$n := $translated ltl constraints
```

Some example translations:

```
#inputs [iA, iB, iC, iD, iE, iF, iG, iH, iI, iJ]
#outputs [oU, oV, oW, oX, oY, oZ]
#1:
((false R (! (iH & (true U iG)) | ((! iD | (! iG U ((oZ & ! iG) & X (! iG U oY))
    )) U iG))))
#2:
((false R (! (iJ & (true U iA)) | (! oX U (iA | ((iI & ! oX) & X (! oX U iB)))))
    ))
#3:
((! (true U oY) | ((! iB | (! oY U (((oX & ! oY) & ! oW) & X ((! oY & ! oW) U oU
    )))) U oY)))
```

Listing 6.2: RERS LTL formulae

```
LTLSPEC NAME rule0 := !(TRUE U (output = 9)) | (!(output = 9) U ((output = 7) &
    !(output = 9) & X(!(output = 9) U (output = 6))))
LTLSPEC NAME rule1 := FALSE V (!((output = 1) & (TRUE U (output = 5))) | (!(
    output = 8) U ((output = 5) | ((output = 7) & !(output = 8) & X(!(output = 8)
     U (output = 12))))))
LTLSPEC NAME rule2 := !(TRUE U (output = 3)) | ((!(output = 1) | (!(output = 3)
    U (!(output = 3) & !(output = 12) & (output = 7) & X((!(output = 3) & !(
    output = 12)) U (output = 10))))) U (output = 3))
```

Listing 6.3: Translated NuSMV LTL formulae

Finally, we concatenate the LTLSPEC lines to the translation of our Mealy machine (listing 6.1), write everything out to a file, and run NuSMV on it. NuSMV will then give us an output like

```
-- specification (!(TRUE U output = 4) | (!(output = 16) U (output = 4 | ((output =
    8 & !(output = 16)) &  X (!(output = 16) U output = 7))))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 6.1 <-
    state = s0
    output = 5
  -> State: 6.2 <-
    state = s0_o_5
    output = 12
    ...
  -- Loop starts here
  -> State: 6.12 <-
    state = s20_o_4
    output = 11
  -> State: 6.13 <-
    state = s20
    output = 4
  -> State: 6.14 <-
    state = s20_o_4
```

```
21      output = 11
```

Listing 6.4: Example NuSMV output

Which we parse into a line like:

```
Rule #8: false  [5;12;10;13;8;14;9;16;4;11;4]([11;4])*
```

6.1.2. NuSMV equivalence checker

Once NuSMV finishes running we parse the output into the infinite paths (or lasso xy^{ω}) they represent. These are essentially normal sequences with an infinite loop at the end. We do not have the ability to pose infinitely long queries to our system, so we need to unroll the infinite loop up to a certain amount of times and ask a finite subset of the infinite query. This is not a perfect method, but better than not checking the counterexamples at all. We define an additional equivalence checking step, the NuSMV checker, seen in 12. This is essentially a simplified variant of the method described in [28] by van de Pol and Meijer.

input : hypothesis H, SUL S, NuSMV counterexamples C, number of unrolls n
output: Counterexample w if one exists, $None$ otherwise

Filter the NuSMV counterexamples C to only keep the input symbols

for $xy^{\omega} \in T$ **do**
  for i *in* $0..n$ **do**
    $w = x \cdot y^n$
    $SULOutput \leftarrow \lambda_S(w)$
    $HypOutput \leftarrow \lambda_H(w)$
    if $SULOutput \neq HypOutput$ **then**
      return w
    end
  end
end
return $None$

Algorithm 12: NuSMV equivalence checker

6.1.3. LTL strategy

Our final approach for solving the LTL problems of RERS 2020 can be summarized as follows:

- Fuzz using libfuzzer
- Learn state machine using TTT with stacked EQ checker containing:
  - smart W-method with horizon of 1
  - corpus eq checker
  - smart W-method with horizon of 3
  - NuSMV checker with unroll of 1000
- Send state machine to NuSMV for final LTL check
- Report results

This allows us to run all 9 problems in under a day on a single machine with a i7-7700k and 32GB ram.

6.1.4. Results

Our method manages to successfully find a correct answer to all the RERS 2019 LTL problems. Note that this is even without the intermediate NuSMV checker, since each problem has 100 LTL formulae to check instead of 2020's 10 which caused it to be too slow. Instead we opted to do a single LTL check at the end of the learning process. For our entry to the 2020 challenge we did use the method mentioned above.

	Time(m:s)	States	Score
Problem 1	00:02	23	100/100
Problem 2	04:26	23	100/100
Problem 3	00:01	24	100/100
Problem 4	00:47	173	100/100
Problem 5	01:33	209	100/100
Problem 6	00:47	183	100/100
Problem 7	37:19	845	100/100
Problem 8	28:01	770	100/100
Problem 9	33:35	810	100/100

Table 6.2: RERS 2019 LTL results

	Time(m:s)	States	Score
Problem 1	02:19	35	10/10
Problem 2	05:26	55	10/10
Problem 3	06:50	107	10/10
Problem 4	07:03	88	10/10
Problem 5	10:03	151	10/10
Problem 6	08:05	96	10/10
Problem 7	49:07	107	10/10
Problem 8	10:13	42	10/10
Problem 9	24:59	77	10/10

Table 6.3: RERS 2020 LTL results

With these results, we ended up winning the sequential LTL track of RERS 2020!

6.2. Reachability

The problems of the reachability track of RERS 2019 and 2020 were of great help in benchmarking and testing our custom state machine learning library, but we had to come to the conclusion that active automata learning is not a great approach in pure reachability contexts where source code is available. For one, active automata learning is at a great disadvantage compared to techniques that are not completely black box, like fuzzing with instrumentation. In this thesis, we looked into how we could leverage fuzzing in order to improve active state machine learning and were succesful in that. However, since the reachability track does not require us to learn a state machine model of the problems in the first place, learning one would just be an extra unneccesary step. Therefore we chose to go with a purely fuzzer-based approach for reachability. To further back up this choice, we compared a simple libFuzzer run to our best performing black box learning based method, TTT combined with the mutating equivalence checker:

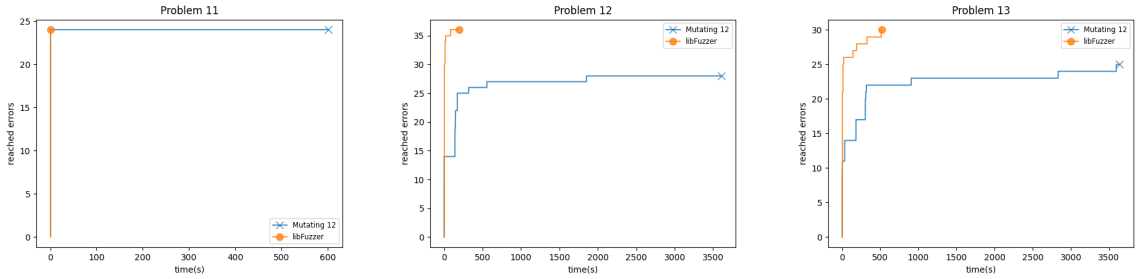

Figure 6.2: Learning vs Fuzzing on the training problems of RERS 2019

Figure 6.2 confirms that fuzzing outperforms our black box learning method, as expected. It finds all reachable errors in a fraction of the time it takes our learner, which does not find all errors within an hour on training Problem 12 and 13.

6.2.1. Reachability strategy

We can be short here, our strategy for the reachability problems goes as follows:

- Modify the RERS code as in 5.1.1
- Run the fuzzers for as long as possible, in our case two months (shorter is probably fine, but we have no way to know in the context of the challenge)
- Run all the files in the corpus as inputs to the unmodified RERS program and record which error states are reached
- Report back with all reached error states

6.2.2. Results

Libfuzzer achieves impressive results on the RERS 2019 reachability track and manages to find every single error state except one in problem 18 within a week of fuzzing. We would have liked to include data on time spent before all states in a problem are reached, but libfuzzer overwrites test cases when it finds a smaller input that reaches the same coverage, which means we cannot rely on the last modified date of files to see when certain errors are reached. We later discovered this and implemented a workaround by using a file watcher to copy new test cases to a folder and keep them if they reach a new error state, but rerunning the fuzzer on all RERS reachability problems would be too time consuming.

	Error states found
Problem 11	20/20
Problem 12	27/27
Problem 13	19/19
Problem 14	24/24
Problem 15	55/55
Problem 16	21/21
Problem 17	41/41
Problem 18	31/32
Problem 19	36/36

Table 6.4: RERS 2019 Reachability results

	Error states found
Problem 11	18/18
Problem 12	17/17
Problem 13	40/43
Problem 14	15/15
Problem 15	54/55
Problem 16	16/16
Problem 17	30/30
Problem 18	39/42
Problem 19	19/19

Table 6.5: RERS 2020 Reachability results

While we did manage to find most reachable error states in RERS 2020, the sequential reachability track also awards points for proving that error states are *un*reachable. Thus, we did not do so well in this track since our method can only reliably determine reachable error states. We could have inflated our score by marking all unreached error states as unreachable, but this would be a guess since we have no way to know if we found all reachable error states. In a verification context this would be unacceptable so we decided not to do this.

7

Conclusions and Future Work

In this chapter we will list our conclusions, discuss potential threats to the validity of our results, and list some of the many avenues for future research.

7.1. Conclusions

Over the course of this thesis, we developed an active state machine learning library in python and used it to participate in the 2020 RERS Challenge. We managed to win the sequential LTL track, but did not do that well in the sequential reachability track since our method can only prove reachability, not unreachability. Furthermore, we managed to answer our research questions:

- **How can we speed up the process of finding counterexamples during equivalence checking?** We managed to develop two supplemental black box equivalence checking methods which, depending on the SUL, greatly speed up equivalence checking during active state machine learning in the minimally adequate teacher framework. The Markov chain based equivalence checker shows promise, but degrades performance in certain cases. The Mutating equivalence checker performs very well in certain cases, finding an order of magnitude more states in the same timeframe. It also does not seem to degrade performance in any of the problems we tested.
- **How can we use white/grey box testing to improve the effectiveness of state machine learning algorithms?** We showed that (when not working in a purely black box scenario) fuzzing is a great technique to use in combination with active learning, but relying on fuzzing alone can be insufficient to fully capture the behavior of a system. This is due to the way fuzzers use their instrumentation to measure coverage and decide what inputs are interesting or not. We also showed that this issue can be alleviated by stacking the fuzz cases with traditional methods such as the W-method. This allows us to harness the speed benefits of fuzzing without sacrificing correctness.
- **The most effective set of techniques for the RERS challenge** We showed that for reachability, using a fuzzer is sufficient to find the reachable error states. Fuzzers also greatly outperform active state machine learning since they leverage instrumentation, while active state machine learning in essence is a purely black box method. In any case, both techniques are unable to prove whether states are *un*reachable (learning is too since we do not know an upper bound on the amount of states), and thus leave some points on the table. In the LTL track, learning state machines has proven to be a more than feasible technique when combined with fuzzing and unrolling LTL counterexamples, as this was good enough get 100% correct answers to the sequential LTL track of RERS 2020 (and 2019 too).

7.2. Threats to Validity

While we did what we could to mitigate any potential threats to validity, we need to acknowledge the following potential problems:

- **Randomness** Several of the techniques used in this work involve randomness. Both the Markov and Mutating equivalence checkers use probabilities to generate new sequences, and thus may have lucky runs where they find states they would normally miss, or the other way around. Because of the time needed to run all experiments, it was deemed infeasible to run them a large amount of times and average the results. However, in most cases, multiple runs were done and no significant difference was observed. Fuzzers also rely on randomness to mutate their corpus and find new coverage, so the same could be said here. Since we fuzzed the problems multiple times on multiple cores, this is also somewhat mitigated.
- **Newly written software** All the learning algorithms, equivalence checkers and more used in this thesis have been implemented from scratch by the author, and thus may contain more bugs or even mistakes than a battle tested library like LearnLib would. Comparing our implementations with LearnLib would have been a good idea, but due to time constraints this could not be done. However, since we participated in the RERS challenge and managed to get a perfect score in the LTL track, which heavily relies on our implementation of the learning algorithms, we can assume that no glaring errors have been overlooked.
- **Limited range of systems to learn** We tested all our code on the RERS 2019 and 2020 LTL and Reachability problems, and found it to work well. However, since these are generated benchmarks, there is no way to tell how well our results would generalize, if they even are applicable, to other settings where active learning is commonly used.

7.3. Future Work

Several interesting directions for future research could be:

- **Markov equivalence checker** We observed that the Markov equivalence checker sometimes generates long sequences of seemingly random inputs when it has insufficient data to calculate the next-item probability for certain n-grams. This could potentially be alleviated by implementing a fallback to shorter n-grams when no data is available for an n-gram of a certain length. Other methods of sequence generation could also be investigated.
- **Pre-clustering** Our attempt at pre-clustering the sequences before using them in the Mutating equivalence checker did not manage to improve performance on the problems we tried it on. Sequence clustering methods other than calculating the longest common subsequence and using that as a distance matrix could be investigated. For example, it seems to be possible to cluster sequences using hidden Markov models [37]. This might work better and allow us to skip the expensive UMAP calculations.
- **Other fuzzers** This may be interesting for future RERS participants: the Angora fuzzer [9] seems very promising and may work even better than AFL and libFuzzer. This definitely seems worthwhile investigating.
- **Fuzzing coverage** There is a discrepancy between states found by active learning, and traces deemed interesting by fuzzers. E.g. not all access sequences leading to unique states are deemed "new coverage" by the fuzzer. This can most likely be explained by the way fuzzer instrumentation works, but this needs to be investigated further. Active learning could maybe even be used as a new way to measure coverage in fuzzers.

A

Figures

A.1. Mutating comparison

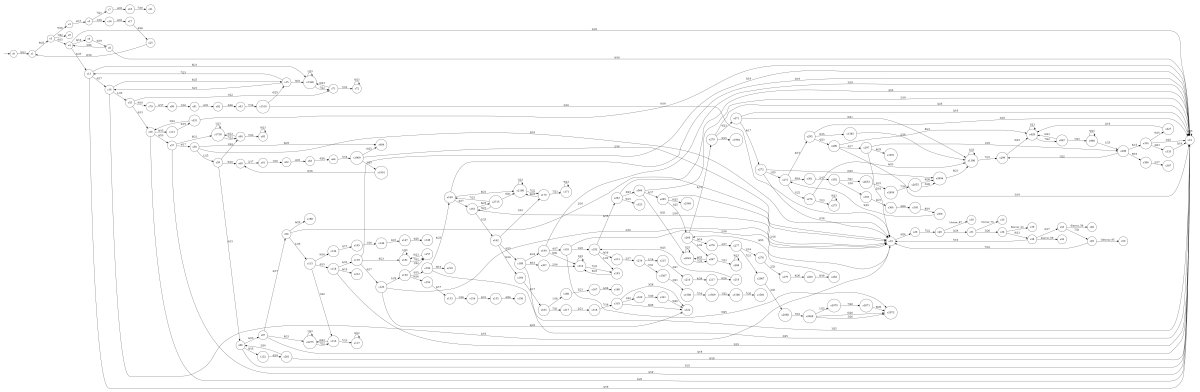

Figure A.1: Part of reachability 2019 problem 12

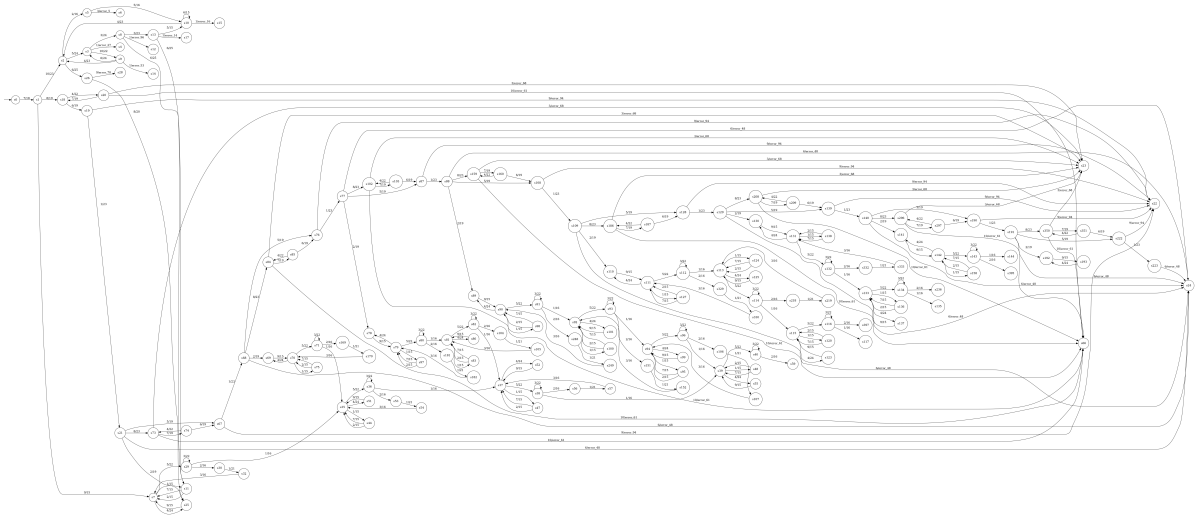

Figure A.2: Part of reachability 2019 problem 13

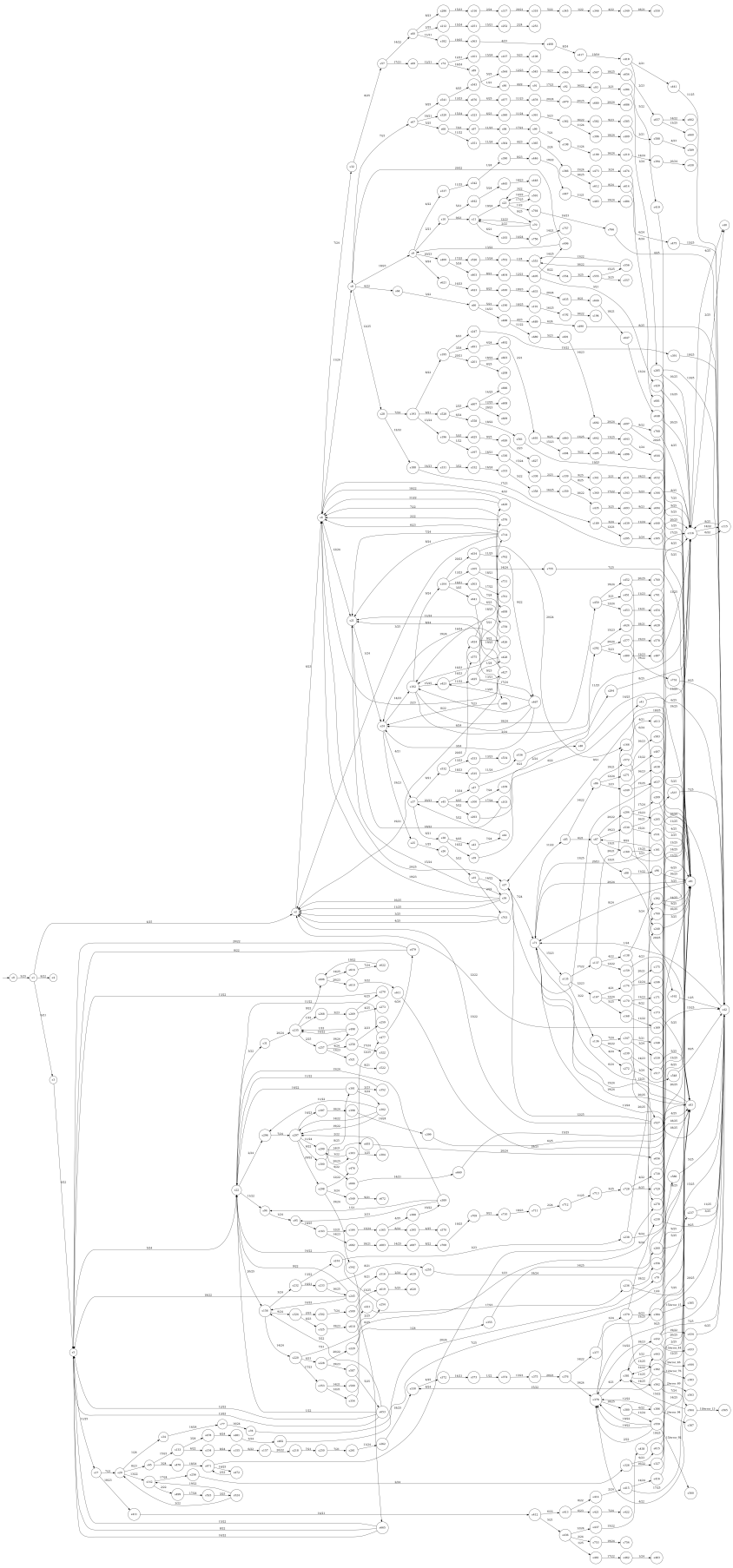

Figure A.3: Part of reachability 2019 problem 18

Figure A.4: Part of reachability 2019 problem 19

Bibliography

[1] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, pages 461–468, 2013. doi: 10.1109/ICSTW.2013.60.

[2] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, 2004.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. ISSN 10902651. doi: 10.1016/0890-5401(87)90052-6.

[4] José L Balcázar, Josep Díaz, Ricard Gavalda, and Osamu Watanabe. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Springer, 1997.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[6] Borzoo Bonakdarpour and Scott A. Smolka. *Runtime verification: 5th international conference, RV 2014 Toronto, ON, Canada, September 22-25, 2014 proceedings*, volume 8734 LNCS. 2014. ISBN 9783319111636. doi: 10.1007/978-3-319-11164-3.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. *8th USENIX Workshop on Offensive Technologies, WOOT 2014*, 2014.

[9] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[10] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978. ISSN 00985589. doi: 10.1109/TSE.1978.231496.

[11] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[12] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. *Proceedings - IEEE Symposium on Security and Privacy*, pages 110–125, 2009. ISSN 10816011. doi: 10.1109/SP.2009.14.

[13] Lesly Ann Daniel, Erik Poll, and Joeri De Ruiter. Inferring OpenVPN State Machines Using Protocol State Fuzzing. *Proceedings - 3rd IEEE European Symposium on Security and Privacy Workshops, EURO S and PW 2018*, pages 11–19, 2018. doi: 10.1109/EuroSPW.2018.00009.

[14] Joeri de Ruiter. A tale of the openSSL state machine: A large-scale black-box analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10014 LNCS:169–184, 2016. ISSN 16113349. doi: 10.1007/978-3-319-47560-8_11.

[15] François Denis. Learning regular languages from simple positive examples. *Machine Learning*, 44(1-2): 37–66, 2001.

[16] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri De Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. *SPIN 2017 - Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, (July):142–151, 2017. doi: 10.1145/3092282.3092289.

[17] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. ISSN 00985589. doi: 10.1109/32.87284.

[18] Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Bas Ploeger, Frank Stappers, Carst Tankink, Yaroslav Usenko, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse, et al. The mcrl2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, page 53, 2008.

[19] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

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

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

[22] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8734:307–322, 2014. ISSN 16113349. doi: 10.1007/978-3-319-11164-3_26.

[23] Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.

[24] Mark Janssen. *Combining Learning with Fuzzing*. PhD thesis.

[25] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: high-performance language-independent model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707. Springer, 2015.

[26] Michael J Kearns, Umesh Virkumar Vazirani, and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[27] Max Kerkers. Assessing the Security of IEC 60870-5-104 Implementations using Automata Learning. (April), 2017.

[28] Tiziana Margaria, Susanne Graf, and Kim G Larsen. *Models, Mindsets, Meta: The What, the How, and the Why Not?: Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200. Springer, 2019.

[29] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *Journal of Open Source Software*, 2(11):205, 2017.

[30] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[31] Edward F Moore et al. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

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

[33] Kostya Serebryany. libfuzzer–a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[35] Rick Smetsers, Joshua Moerman, and David N. Jansen. Minimal separating sequences for all pairs of states. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9618:181–193, 2016. ISSN 16113349. doi: 10.1007/978-3-319-30000-9_14.

[36] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing model learning with mutation-based fuzzing. *arXiv preprint arXiv:1611.02429*, 2016.

[37] Padhraic Smyth. Clustering sequences with hidden markov models. In *Advances in neural information processing systems*, pages 648–654, 1997.

[38] Bernhard Steffen. An Abstract Framework for Counterexample Analysis in Active Automata Learning. *JMLR: Workshop and Conference Proceedings*, (1993):79–93, 2014.

[39] Hasan Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992. ISSN 01403664. doi: 10.1016/0140-3664(92)90092-S.

[40] Wesley Van Der Lee. Vulnerability Detection in Mobile Applications Using State Machine Modeling.

[41] Nan Yang, Kousar Aslam, Ramon Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. Improving Model Inference in Industry by Combining Active and Passive Learning. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pages 253–263, 2019. doi: 10.1109/SANER.2019.8668007.