Felix Wallner[1], BSc

Development of a Robust Active Automata Learning Algorithm for Automotive Measurement Devices Avoiding Resets

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard K. Aichernig

Institute of Software Technology (IST)

Graz, May 19, 2022

[1] E-mail: felix.wallner@ist.tugraz.at

AFFIDAVIT

Abstract

Testing and verifying industrial devices is often an elaborate process, as such systems are frequently restricted to certain locations and their operating time is valuable. Therefore, the efficiency and fast generation of behavioral models is of great importance. With model learning we can learn a system's behavior automatically. The learned models can then easily be used for system verification instead.

This thesis presents an active automata learning algorithm for learning Moore machines. Differently from commonly used active learning techniques, the presented algorithm does not require the system to be reset during learning. We want to avoid the usage of resets, as they might be costly in terms of time for the kind of physical devices we aim to learn. Additionally, we exploit the structure of Moore machines in our algorithm to perform fewer input sequences to distinguish states. To learn the models of industrial measurement devices, we extend our algorithm to handle systems with time-triggered transitions by only learning partial models. This prevents the occurrence of non-deterministic behavior due to race conditions during execution.

In our evaluation we show that our algorithm can learn most strongly-connected models, but that its success rate and performance is strongly correlated to the number of unique outputs of the system we want to learn. A limitation of the algorithm is that in the case where the system has no unique outputs, our algorithm is unable to learn it entirely. The evaluation includes test runs of our algorithm on over 75 000 different, randomly generated, Moore machines with different combinations of number of states, number of unique outputs, and algorithm parameters. The results show that the algorithm learns small to medium sized automata with a high number of unique outputs well. We also learn a majority of the other combinations, however, with a lower number of unique outputs it suffers from outliers that decrease its consistency overall.

We show in our case study that the algorithm successfully learns the behavior of a measurement device that is used in the automotive industry. We also learn a model of an existing digital twin, i.e., a virtual behavioral copy, of said device. We manage to find differences between both models of the devices and use our automatically learned model of the real device to improve the digital twin.

Keywords: Active Automata Learning, Reset-less Learning, Moore Machines, Model Learning, Conformance Testing, AALpy.

Kurzfassung

Das Testen und Prüfen von industriellen Geräten ist oft ein aufwendiger Prozess, da diese Geräte häufig an einen gewissen Standort gebunden sind oder ihre Verwendung teuer ist. Die effiziente und schnelle Erzeugung von Modellen ist deshalb von großer Wichtigkeit. Mit Modell-Lernen können diese automatisch kreiert und für Systemprüfungen eingesetzt werden.

Diese These präsentiert einen aktiven Automaten-Lernansatz, der Moore Maschinen ohne Resets lernen kann. Wir wollen die Verwendung von Resets vermeiden, da sie potentiell teuer im Sinne des Zeitaufwandes für die Art von Geräten, die wir lernen wollen, sein können. Zusätzlich wollen wir die Struktur von Moore Maschinen für unseren Algorithmus ausnutzen, um weniger Inputfolgen für die Unterscheidung von Zuständen verwenden zu müssen. Um die Modelle von industriellen Messgeräten zu lernen, erweitern wir unseren Algorithmus sodass er auch mit zeit-ausgelösten Zustandsübergänge zurechtkommt, indem wir nur partielle Modelle lernen. Dies verhindert das Auftreten von nichtdeterministischen Verhalten durch Wettlaufsituationen während der Ausführung des Algorithmus.

In unserer Auswertung zeigen wir, dass unser Algorithmus die meisten stark zusammenhängende Modelle lernen kann, dass aber seine Erfolgswahrscheinlichkeit und seine Leistung mit der Anzahl an eindeutigen Ausgabewerten des zu lernenden Systems zusammenhängt. Je mehr eindeutige Ausgabewerte das System beinhaltet, desto besser kann es gelernt werden. Eine Einschränkung unseres Algorithmus ist, dass in Fällen in denen ein System keine eindeutigen Ausgabewerte hat, der Algorithmus nicht in der Lage ist das Modell zu lernen. Unsere Auswertung umfasst Testläufe unseres Algorithmus auf über 75 000 verschiedenen, zufällig generierten Moore Maschinen mit unterschiedlichen Kombinationen aus der Anzahl von Zuständen und eindeutigen Ausgabewerten, sowie die Wahl unterschiedlicher Werte für weitere Parameter. Diese Resultate zeigen, dass der Algorithmus in der Lage ist kleine bis mittelgroße Automaten mit hoher Anzahl an eindeutigen Ausgabewerten gut zu lernen, dass er aber mit geringerer Anzahl selbiger eine beträchtliche Anzahl an Ausreißer aufzeigt und dadurch seine Verlässlichkeit sinkt.

Wir zeigen in unserer Fallstudie, dass der Algorithmus unser Fahrzeug-Messgerät erfolgreich unter praktischen Bedienungen lernt. Er lernt auch einen bereits existierenden Digitalen Zwilling, eine virtuelle Kopie unseres Messgeräts. Wir finden Unterschiede zwischen den beiden Geräten und verwenden unser automatisch gelerntes Modell des echten Gerätes um den Digitalen Zwilling zu verbessern.

Schlagworte: Aktives Automaten-Lernen, Reset-freihes Lernen, Moore Machinen, Modell-Lernen, Konformitätstesten, AALpy.

Acknowledgements

I would like to thank everyone who accompanied me during my studies, who supported and motivated me, and who made this part of my life an exciting and educational time. Especially, I want to thank those with whom I shared interesting conversations and amazing times.

A special thank you to my supervisor Bernhard K. Aichernig, who initially aroused my interest in the topics of automata learning and formal methods. I would like to thank him for the opportunity of allowing me to initially join his team as a tutor, teaching one of my favorite subjects during my entire studies, Logic and Logical Programming. I also want to thank him for his comprehensive support for first my Bachelor's thesis and now this one. I would like to thank him for the opportunity of working in the LearnTwins project and our thought provoking discussions.

A big thanks goes to Christian Burghard, who not only provided the inspiration for the algorithm, but who made the time for brainstorming, checking and applying the algorithm in practice. I would like to thank him for always answering my questions promptly and having the patience to help me throughout the entire development of the algorithm, from start to finish.

I would also like to thank Martin Tappler for his excellent support on my scientific journey. His backing and guidance on the topic of my Bachelor's thesis led to the release of our research on the International Conference on Tests and Proofs (TAP), which was a big achievement as my first publication. Without this success I might have turned my attention to other topics and this thesis would probably not exist in this form without him.

Furthermore, I would like to thank both Martin Tappler and Andrea Pferscher for being always open for answering my questions and giving explanations to a wide variety of topics, which helped me immensely during the writing of this thesis.

Special thanks to my parents, Ulrike and Dieter Wallner, who always supported me in my studies, even though they still believe this thesis deals exclusively with the learning of coffee-machines, and made it possible for me to study this way. I would like to thank my brother, Andreas Wallner, who always supported me and for proof-reading this thesis. Finally, I would like to give an especially big thank you to my grandmother, Doris Schwarz. She always supported me, but also pushed me to achieve my best in all aspects of life but notably in my studies and education.

This work was funded by the LearnTwins project (No 880852) from the Austrian Research Promotion Agency (FFG).

Felix Wallner
Graz, Austria, May 19, 2022

Danksagung

Ich möchte mich herzlich bei allen bedanken, die mich über mein Studium hinweg begleitet haben, die mich unterstützt und motiviert haben und mein Leben in dieser Zeit spannend und lehrreich mitgestaltet haben.

Ein besonderer Dank gilt meinem Betreuer Bernhard K. Aichernig, der mein Interesse für das Gebiet des Automaten-Lernen und der Formalen Methoden geweckt hat. Ich möchte ihm für die Gelegenheit danken, dass ich seinem Team ursprünglich als Tutor beitreten durfte, um eines meiner Lieblingsfächer während meines Studiums zu unterrichten, Logik und Logisches Programmieren. Ebenfalls möchte ich ihm für seine umfassende Unterstützung danken, zuerst während meiner Bachelorarbeit und nun auch während dieser Masterarbeit. Ich bedanke mich für die Gelegenheit am LearnTwins Projekt teilnehmen zu dürfen und für die interessanten Gespräche, die wir geführt haben.

Großer Dank geht auch an Christian Burghard, der nicht nur die ursprüngliche Inspiration für den Algorithmus geliefert hat, sondern sich auch Zeit für das Brainstorming sowie die Überprüfung und Anwendung des Algorithmus genommen hat. Ich möchte ihm danken, dass er immer meine Fragen schnell beantwortete und sehr geduldig über den gesamtem Entwicklungsprozess hinweg war.

Ich möchte mich auch bei Martin Tappler für seine ausgezeichnete Unterstützung auf meiner wissenschaftlichen Reise bedanken. Seine Betreuung und Beratung für meine Bachelorarbeit haben zu der Veröffentlichung unserer Forschungsergebnisse auf der International Conference on Tests and Proofs (TAP) geführt, was als meine erste Publikation ein großer Erfolg für mich war. Ohne diesen Schritt hätte ich mich vielleicht anderen Themen zugewendet und diese Arbeit würde in dieser Form wahrscheinlich nicht existieren.

Weiters möchte ich sowohl Martin Tappler als auch Andrea Pferscher dafür danken, dass sie mir immer offene Fragen beantwortet oder Konzepte erklärt haben, welche mir für das Verfassen dieser Arbeit ungemein geholfen haben.

Ein spezielles Dankeschön an meine Eltern, Ulrike und Dieter Wallner, die mich immer in meinem Studium unterstützt haben, auch wenn sie noch immer glauben, dass meine Arbeit ausschließlich aus dem Lernen von Kaffeemaschinen besteht, und die mir mein Studium in dieser Form ermöglicht haben. Ich möchte mich bei meinem Bruder, Andreas Wallner, bedanken, der mich immer unterstützt hat und für das Korrekturlesen dieser Arbeit. Zuletzt möchte ich ein großes Dankeschön meiner Großmutter, Doris Schwarz, aussprechen. Sie hat mich nicht nur immer in allem unterstützt, sondern mich auch immer angespornt neue Leistungen in meinem Leben zu erbringen, besonders aber im Bereich meines Studiums und meiner Ausbildung.

Diese Arbeit wurde vom LearnTwins Projekt (No 880852) der Österreichische Forschungsförderungsgesellschaft (FFG) finanziert.

Felix Wallner
Graz, Österreich, 19.5.2022

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

DFA	Deterministic Finite Automata
DT	Digital Twin
FSM	Finite State Machine
MAT	Minimally Adequate Teacher
QA	Quality Assurance
SUL	System Under Learning

1 Introduction

1.1 Motivation

In our modern world new software systems are developed rapidly with ever increasing requirements and demands on their functionality, which results in more and more complex software systems. Especially reliability of these systems is of great importance. To this effect, the field of quality assurance (QA) is responsible for the validation and verification of such systems, however, the testing and maintenance of such systems with ever increasing complexity is difficult and time consuming.

Not only does this affect software systems, but also physical devices. Testing the latter is even more elaborate, as the devices are often confined to a certain location and are expensive to use. Time that is spent testing or verifying a physical device is time that is not spent in production, which may be costly. Especially, the automotive industry uses a large number of measuring devices that have the aforementioned properties. It is therefore desirable to acquire a virtual copy of the physical device, which is called digital twin (DT), that mirrors the behavior of the device as close as possible. Working on DTs for QA avoids most of the downsides of working on the physical device, as described above, if the DT is close enough to the real system.

DTs of such devices can be created by hand, based on the instruction manual or personal experience with the device, however, this method does not only require the effort of qualified personnel but is also error-prone, if the manual is incorrect or the device has undocumented behavior, but may actually sometimes be impossible, if no instruction manual or high-level code exists of the device. Therefore, a more sophisticated approach to gain insight into the logic and workings of such devices is the automatic exploration and learning of the black-box device from the outside using intelligent trying of inputs and observing of outputs.

Pelet et al. [31] were among the first to present a method to infer models from black-box systems using active automata learning. Partially due to their work, this developed into an active area of research in the field of QA in which many different approaches are created and improved upon as shown in a recent survey by Aichernig et al. [4]. One solution that supplies methods and tools in learning models in such a way is provided by the open-source Python [17] framework AALpy [28]. Among others it provides an implementation of common learning algorithms, such as L^* by Angulin [10], which is one of the most established learning algorithms introduced and, although created in the 1980s, is still a staple in active automata learning today.

A caveat to most commonly used active automata learning algorithms is that they require the ability to *reset* a system. Resetting a system means bringing it back into a known state on which the learning observations can be based. However, some systems are difficult or time consuming to reset, e.g., the Smoke Meter [11] device contains a heater that would require a substantial amount of time to turn off and on again. Considering that such resets would be required in large numbers, this approach becomes infeasible.

Instead, we want to use an approach that does not require the resetting of the system in order to learn it. While a number of algorithms that work reset-less do exist, such as the one by Rivest and Schapire [34] or by Groz et al. [19], these work on deterministic finite automata (DFAs) and Mealy machines respectively. One of the advantages of the industrial measurement devices is that they are not fully black-box systems. They have some internal state variables and sensor values which we can access using queries. We want to take advantage of these and model our devices as Moore machines instead, as they represent the capabilities of these devices better. Therefore, we develop our own algorithm that works reset-less while exploiting the structure of Moore machines to learn the models of systems such as the automotive industrial measurement devices.

1.2 Research Problem and Goals

The goal is to develop a new active automata learning algorithm that works without resets and to adapt it to specifically learn the models of measurement devices from the automotive industry such as the ones described by Aichernig et al. [3]. Especially the reset-less nature of the algorithm was an important requirement, as the industrial measurement devices may not easily be resettable or may require an impractically long time to do so.

In literature there are a variety of active automata learning algorithms, such as, most prominently, Angluin's L^* algorithm [10] and other learning approaches such as the TTT algorithm by Isberner et al. [21] or the KV algorithm by Kearns and Vazirani [24]. However, all of these require the resetting of the system and while reset-less algorithms do exist, such as the one by Rivest and Schapire [34] or the hW-inference algorithm by Groz et al. [19], we wanted to specifically take advantage of the structure of Moore machines, which our industrial measurement devices were modeled as.

Specifically, the *AVL 415SE Smoke Meter* [11] was chosen as a representative for the type of measurement devices we wanted to learn and presented additional challenges in the form of time-triggered behavior that we solved with an extension to the base algorithm. We also wanted to compare our learned model of the Smoke Meter with an already existing DT to ascertain if we could detect differences in behavior between them and to potentially enhance the DT with information that we learned from the real device.

Additionally, we wanted to create a concrete implementation of the algorithm [39] that could also be used on any number of other measurement devices from the automotive industry. The implementation should be created in an established framework to allow the use of a wide variety of testing algorithms and to take advantage of existing infrastructure developed for this framework. We chose AALpy [28] as the framework and implemented our algorithm and its extension for time-triggered behavior in Python [17].

Finally, the algorithm should be evaluated in its successful learning rate and its general performance. We did this by testing the algorithm on over 75 000 different randomly generated Moore machines and aggregating the resulting metrics, which we analyzed in-depth.

1.3 Context of this Thesis

The algorithm and methods presented in this thesis were developed as part of the LearnTwins[1] project from the Austrian Research Promotion Agency (FFG), a collaboration between *AVL List GesmbH*, *AIT Austrian Institute of Technology GmbH* and *Graz University of Technology*. The project goal is the learning of DTs for validation, verification, and testing of cyber-physical systems through the development of tools and methods to enable these processes.

The LearnTwins project aims to create trustable DTs, which should be correct, reliable and cost-effective. Hence, they should be created automatically via learning methods. To accomplish this goal the project aims to combine a number of different methods such as automata learning, classical machine learning, and deep learning among others.

As part of this effort, and to specifically learn automotive industrial measurement devices such as the Smoke Meter presented in Section 6.2, this algorithm and its extension for time-triggered behavior was developed. This thesis was written to document the algorithm, analyze its performance and present a comprehensive review of its workings and results. It provides a formal framework for the algorithm and represents the basis for future improvements and extensions to this algorithm in creating new, and further enhancing existing, DTs.

[1]Information about the project and the publications created as part of it can be found on the project website: `https://learntwins.ist.tugraz.at`

1.4 Structure

This thesis is structured as follows:

Chapter 2 consists of the preliminaries for this thesis. It introduces concepts such as active automata learning and Moore machines, which our algorithm will learn, as well as our modified adequate teacher framework and the use of resets, or the lack thereof, in this thesis.

The methodology and the base algorithm is presented in Chapter 3. We define the data structures and methods our algorithm uses to learn Moore machines reset-less. Additionally, we also explain the limitations of this approach.

In Chapter 4 we then show two illustrative examples in depth, detailing the workings of the algorithm and present the intermediate results to demonstrate the base algorithm from the previous chapter.

Chapter 5 presents an extension for the base algorithm that we required to deal with the time-triggered behavior that the industrial measurement devices exhibit. This behavior would otherwise introduce non-determinism into the learning process, which the base algorithm cannot handle. We demonstrate what exactly this behavior is, why it occurs and how we elected to enhance the base algorithm to work with it.

Chapter 6 contains the evaluation of the base algorithm, showing its success rate and performance, as well as the case study performed on the Smoke Meter industrial measurement device, which we learned successfully. Additionally, we compare the learned model of the Smoke Meter industrial measurement device to an existing digital twin of said device and show that we can find differences in behavior between them, which we then used to improve the DT.

Finally, in Chapter 7 we discuss related work. We also state future work, potential directions in which the algorithm could be extended as well as enhancement that could be added in the future. We present a summary of this thesis and close with a conclusion about our work.

2 Preliminaries

2.1 Active Automata Learning

Automata learning is the act of deducing the behavior and logic of a system based on observations. We distinguish between two variants of automata learning: passive and active.

In the *passive* approach the system is learned from a given set of execution traces, records of observations about single executions of the system. Based on these execution traces, the observed inputs and outputs, the system can be modeled if enough such traces exist and if they cover every aspect of the systems behavior. Due to the execution traces already existing no further access to the system is required during the learning. In fact, the generation of execution traces and the learning process are completely independent.

In contrast, *activate* automata learning provides the inputs to the system directly and may generate new learning data as required. These questions are asked of an oracle that can correctly answer certain questions or perform operations on the system usually in real time. This approach requires less data in advance as knowledge is retrieved on demand and the learning algorithm may only generate the data that is actually required to model the system, thus often requiring fewer tests. One of the first active learning algorithms, L^*, was introduced by Angluin [10] and is still consulted for many modern activate learning methods.

A black-box system is one where the assumptions about its behavior may *only* stem from the inputs applied to the system and the outputs observed to them. The system is considered "opaque" to the observer as one cannot look inside the "black-box" to see its internal workings thus requiring the act of learning its behavior in the first place. In contrast to a black-box system would be a white-box system where the internal structure of the system is known in advance such as the source code. There is generally more interest in testing white-box systems, as described by [29], but not so much in learning them, as their behavior is already known, so they are of no further interest to this thesis. Learning the behavior of a black-box system requires no domain specific knowledge and can be based exclusively on its observed outputs. Only the input alphabet must be known in advance. Everything else, such as outputs, can be observed during the learning process.

2.2 Minimally Adequate Teacher Framework without Resets

The Minimally Adequate Teacher (MAT) framework first introduced by Angluin [10] is a system in which a learner may learn a black-box system from a teacher. The teacher communicates with the *system under learning* (SUL) directly while the learner only communicates with the teacher using queries. The learner uses the information from these queries to form and then refine a so called *hypothesis*, a model of the internal behavior and logic of the SUL, which might not be complete yet as some behavior might not have been observed or incorporated yet.

In Figure 2.1 we present the modified version of the MAT used in this thesis. The two major changes compared to the classical version of the MAT are that the output oracle cannot and is not required to reset the SUL after each *output query* and that the learner may observe all *output queries* performed during *equivalence queries*.

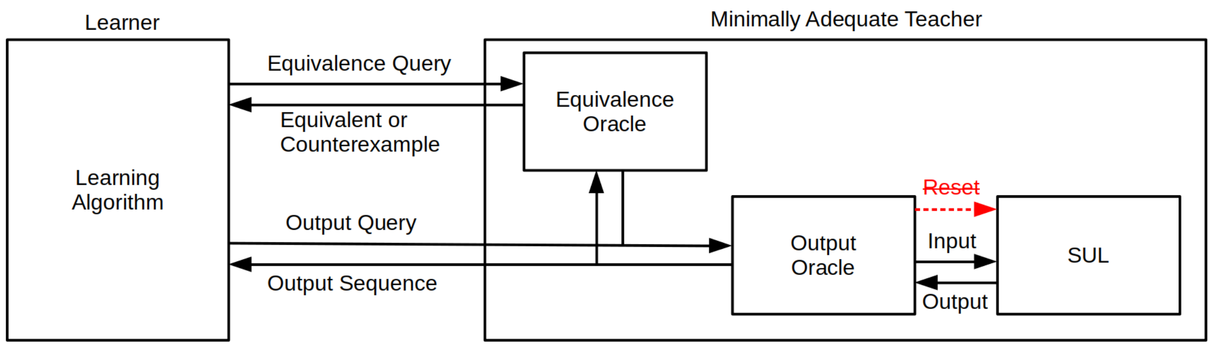

Figure 2.1: *Minimally Adequate Teacher* (MAT) framework without automatic resets from output oracle.

Output Query

A question about which sequence of outputs corresponds to a given sequence of inputs. In the classical framework these types of queries are also called *membership queries* and can be used to determine if a given sequence of inputs is part of the language of the SUL. This is generally done by providing an input word to the teacher who then answers with a response sequence of equal length composed of the outputs of the SUL. These queries, the provided inputs, and their corresponding outputs, may then be used by the learner to form a *hypothesis* about the structure of the SUL.

Normally, the teacher *resets* the SUL before executing an input sequence on the SUL either by providing a *reset sequence*, a word of inputs known in advance, or by performing a special procedure, like opening a new connection or restarting a device, that will always result in bringing the SUL into the same known state. In this thesis, as will be explained in Section 2.3, we do not know any such reset sequence in advance, nor can we use a special procedure to reset the SUL. Therefore, we do not require an output query to start at the same state but instead perform it from the state the SUL is currently in.

Equivalence Query

A question about whether or not a given hypothesis provided by the learner is equivalent to the SUL. The teacher may conclude that the provided hypothesis is equivalent or alternatively provides a counterexample composed of some inputs and outputs of the SUL which shows a difference in behavior compared to the hypothesis. The counterexample may then be used to improve the hypothesis further.

In the case of the SUL being a black-box, general equivalence cannot be checked by the teacher. Instead a so called *equivalence oracle* may resolve such queries by checking for conformance, whether the SUL conforms to the hypothesis, via a series of output queries.

It is important to note that in our setup of the MAT in Figure 2.1 the learner may observe all output queries that the equivalence oracle performs. It is not sufficient to only receive a counterexample because all intermediate inputs and outputs will be used by the learner in addition to any counterexample to enhance the hypothesis. More formally, we require that the equivalence oracle answers with the full trace of inputs and outputs that it performed on the SUL which is equivalent to observing the output queries.

As the output oracle cannot directly reset the SUL, the equivalence oracle must compute a sequence of inputs to navigate the SUL to a chosen starting state of the hypothesis using an output query every time a classical membership query should be performed during the equivalence checking.

2.3 Use of Resets

Most active automata learning algorithms assume the existence of, and depend on, the ability to reliably *reset* a SUL, i.e., bringing the SUL into its initial state. The queries described in the classical MAT and the learning algorithms using it depend on such a reset, thus allowing them to process all observations in relation to this static, initial known state of the SUL.

However, the ability to reset a SUL is not always given or might be very expensive. For example, resetting an industrial machine may require long startup times because the device requires certain operating temperatures or other resources. In some cases it might actually be impossible to reset the SUL.

For this reason the development of reset-less activate automata learning algorithms is of interest especially in the case of industrial automata devices as the time needed to reset them could be immense.

Working from within the MAT comes with a number of advantages, most importantly the usage of established *testing algorithms* in equivalence oracles such as the W-method by Chow [16] that comes with a number of very useful guarantees such as the guarantee to find any sequencing errors up to a bounded number of states. We would still like to use these established procedures, however, as already stated, we cannot use resets in the classical way. Instead, when the equivalence oracle wants to reset the SUL, we will navigate to a chosen starting state within our strongly-connected component using our best knowledge of the current hypothesis. As we will only ever perform an equivalence query while our hypothesis is deterministic (see 3.3.4) detecting a discrepancy during a reset sequence while performing an equivalence query can be used directly as a counterexample. Aside from these reset sequences the testing algorithm can proceed as normal.

Outside of equivalence queries the learner will operate from the current state or navigate to states that it can reach using output queries as described above, i.e., without performing resets between output queries. Most of these inputs are performed step-wise with output queries of length one, as the learner adapts to unexpected behavior from the SUL. Classical membership queries (with reset) are not required during normal learning and potentially not even possible if the starting state is either not reachable or the hypothesis is non-deterministic.

2.4 Moore Machines

Moore machines as first described by Moore [27] are a type of Finite State Machine (FSM) which can classically be defined as a 6-tuple, where the Moore machine M is:

$$M = (I, O, S, s_0, \delta, \lambda) \tag{2.1}$$

where

- I is the finite, non-empty set of input symbols
- O is the finite, non-empty set of output symbols
- S is the finite, non-empty set of states
- s_0 is the initial state $\in S$
- $\delta : S \times I \to S$ is the state transition function
- $\lambda : S \to O$ is the output function.

An example of a Moore machine can be seen in Figure 2.2.

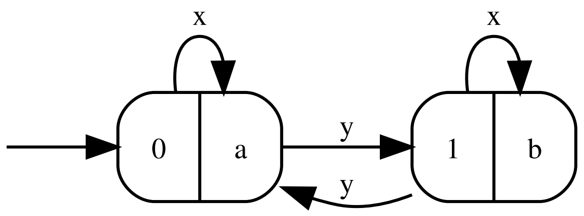

Figure 2.2: Example of a Moore machine.

Drawing Convention of Moore Machines. In this thesis, Moore machines are depicted with unique state numbers on the left and output on the right of every state. The transitions are labeled with the corresponding inputs. The unique state numbers are chosen based on the order in which the states were created, i.e., state 0 would be the very first state created. This convention will be useful later when states will be split, merged and deleted, to keep track of the order they were created in.

Formal Notation of Sequences. For a given input sequence $\alpha x \in I^*$, where $\alpha \in I^*, x \in I$ and $\alpha \cdot x = \alpha x$, and a state $s \in S$ we define the notation for sequences similar to Giantamidis et al. [18]: For $\delta^* : S \times I^* \to S$ we define $\delta^*(s, \epsilon) = s$ as well as $\delta^*(s, \alpha x) = \delta(\delta^*(s, \alpha), x)$. We furthermore define the notation for $\lambda^* : S \times I^* \to O^+$ as $\lambda^*(s, \epsilon) = \lambda(s)$ and $\lambda^*(s, \alpha x) = \lambda^*(\alpha)\lambda(\delta^*(s, \alpha x))$.

Transfer Sequence. A *transfer sequence* is an input sequence $\alpha \in I^*$ which, according to δ, leads from a source state $s \in S$ to a target state $s' \in S$, i.e., $\delta^*(s, \alpha) = s'$. We also call transfer sequences *words* for short.

Most commonly used learning algorithms work on Mealy machines instead of Moore machines. The difference between Mealy and Moore machines is that output of the former depends on the current state *and* a give input, while the latter only depends on the current state. Formally, the output function of a Mealy machine is $\lambda : S \times I \to O$ and is otherwise identical to the Moore machine. Furthermore, there does exist a general method for transforming a Moore machine into a Mealy machine and vice verse, i.e., it is possible to transform a Moore machine into a Mealy machine of equal size while a transformation of a Mealy machine into a Moore machine might increase its number of states in the process which was explored in detail by Klimovich and Solov'ev [25].

We chose to model the industrial measurement devices, which we are interested in learning, as Moore machines instead of Mealy machines, as their behavior is closer to the former than the latter. We explain this in detail in Section 6.2.1, but in short, we have access to some of the internal state variables of the industrial measurement devices, which may be queried at any time. Because we may query this internal state we model the SUL as Moore machines instead, which we take advantage of in our algorithm.

2.5 Reset-less Learning

Normally, a reset would bring a system into the same known state so the learner could never get "lost" and could always orient itself by said starting state. Not having access to such a reset also means not being able to "anchor" our observations to a known starting state. One of the greatest challenges in reset-less learning, therefore, is to identify the state the SUL is currently in to correctly infer state transitions and states from the produced outputs to input queries.

In identifying states using a certain sequence of inputs and observing the corresponding outputs there are three types of special sequences which we differentiate:

Homing Sequence. A *homing sequence* is a sequence of inputs $h \in I^*$ that, when applied to the SUL, uniquely determines the reached stated by the produced outputs, that is, the reached state *after* the homing sequence is performed. Formally, h is a homing sequence iff

$$\forall s, s' \in S : \lambda^*(s,h) = \lambda^*(s',h) \rightarrow \delta^*(s,h) = \delta^*(s',h) \tag{2.2}$$

A homing sequence is similar to a reset sequence as the state after it is applied is known, in contrast to such, however, the end state is not always the same.

Distinguishing Sequence. In contrast to a homing sequence a *distinguishing sequence* $d \in I^*$ uniquely determines the *starting* state from which a distinguishing sequence is applied instead of the end state, by producing a different sequence of outputs for every starting state. Formally, d is a distinguishing sequence iff

$$\forall s, s' \in S : s \neq s' \rightarrow \lambda^*(s,d) \neq \lambda^*(s',d) \tag{2.3}$$

It should be noted that every distinguishing sequence is necessarily a homing sequence in a deterministic machine as identifying the starting state before a given input sequence also determines the end state, however the inverse is not true.

Reset Sequence. Finally, a *reset sequence* $r \in I^*$, which is also called a *synchronizing sequence* in Rivest and Schapire [34] and others, brings the SUL into the same known state after it is applied to the SUL. Formally, r is a reset sequence iff

$$\exists! s' \in S, \forall s \in S : \delta^*(s,r) = s' \tag{2.4}$$

Every reset sequence is necessarily a homing sequence while the inverse is not true.

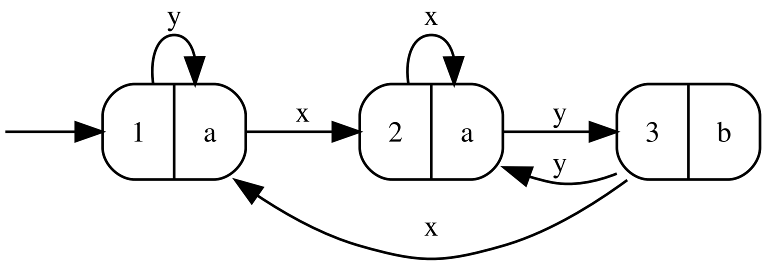

Figure 2.3: Example Moore machine for sequences in Table 2.1.

Table 2.1: Output sequences $\lambda^*(s,\alpha)$ from given starting state s and input sequence α for Moore machine shown in Figure 2.3.

s \ α	ϵ	x	y	xx
1	a	aa	aa	aaa
2	a	aa	ab	aaa
3	b	ba	ba	baa

To better illustrate the different types of sequences discussed above we provide some examples in Table 2.1 corresponding to the Moore machine in Figure 2.3. The table contains the output sequences $\lambda^*(s,\alpha)$ given the starting state $s \in S$ and an input sequence $\alpha \in I^*$. In this case, x is a homing sequence, y a distinguishing sequence and xx a reset sequence:

The sequence x is homing, as applying x and observing aa is guaranteed to end in state 2 while observing ba ends in state 1. It is however neither distinguishing, as observing aa might have started in states 1 or 2, nor a reset, as there are different end states depending on the observed outputs.

The sequence y is distinguishing as applying it to each state results in a unique output sequence.

Finally, applying sequence xx is guaranteed to end in state 2 and is thus both a homing sequence and a reset sequence.

Kohavi and Jha [26] explain the different types of sequences in more detail in addition to differentiating between *preset* and *adaptive* versions of each: A *preset* sequence is one known in advance which can be applied in one single query while an *adaptive* sequence changes based on the observed outputs and may therefore be executed in multiple queries or steps.

Generally, every FSM always has a homing sequence, however, there is no guarantee that an FSM has a distinguishing sequence. If none exists, then identifying a starting state may not be possible by providing a single input word. Instead, a state may be identified by applying multiple input sequences to said state until the set of corresponding outputs is unique:

Characterization Set. We follow the definitions of Groz et al. [20], namely, that two states $s, s' \in S$ are distinguishable by the input sequence $\alpha \in I^*$ if $\lambda^*(s, \alpha) \neq \lambda^*(s', \alpha)$ and that they are distinguishable by a set $W \subset I^*$ if there exists such an $\alpha \in W$. This set W, commonly called a W-set as introduced by [38], is the *characterization set* if W distinguishes every pair of states in an FSM. Mapping every input sequence of such a W-set to its corresponding output sequences for a given state is called its *state characterization* and can be used to uniquely identify any state. By definition, every state characterization must be different between each pair of states in an FSM similar to how states can be uniquely determined by the output of the execution of a distinguishing sequence. An FSM for which all states can be distinguished from each other in this form is called *minimal*.

Finally, an important tool for reset-less algorithms is the information about previous queries and sequences of observed outputs given certain inputs. To keep track of these and the current state of the SUL we use the following:

Global Trace. Most reset-less learning algorithms, ours included, maintain a sequence of inputs and their corresponding outputs from the beginning of the learning procedure. This sequence, the *global trace* or just the trace, is continuously updated with all inputs/outputs that the SUL is queried with. It represents the entire history of queries to the SUL and can be used to compare behavior at different points during learning or, in our case, to infer distinguishing sequences between states with the same output. In our case the global trace includes the inputs/outputs of all queries including the ones performed by the equivalence oracle as described in Section 2.2.

3 Methodology and Algorithm

3.1 Overview

Assumptions. Some core assumptions are required to meaningfully learn the behavior of a system using our method of active automata learning:

- The system behaves like a Moore machine, i.e., its current output is only determined by its internal state and not by any other factors.
- The input alphabet is known
- The system is input-complete. It must always accept any input from the input alphabet and cannot refuse one.
- The system is deterministic.
- The system has a finite number of states.
- The system cannot be reset using an external mechanism as stipulated by the classical MAT by Angluin [10] as explained in Section 2.2. A reset may still be simulated by navigating to a certain chosen starting state using a *reset sequence* as described in Section 2.5.

While it is not necessarily a core assumption for the algorithm itself, the nature of reset-less learning allows us only to learn *strongly-connected* components of the system because there is potentially no way to reach a state again after we left it if there are no transitions back to it. In the case of a system that is not strongly-connected the algorithm would only learn a strongly-connected *subset* of the system. In that case the algorithm would not fail due to the required robust nature however. If learning the *entire* system is the goal then we postulate the additional requirement that the underlying system is *strongly-connected.*

Task Definition. Our task can be described as:

With a given system under the assumptions above, find a minimal Moore machine to which the system conforms, i.e., the system and the model should produce the same input/output sequences, if the starting state is known.

The reason we are not able to guarantee that the starting state of the Moore machine will be identified goes back to the explanation in Section 2.5, namely that there might not exist a distinguishing sequence for any given Moore machine. Additionally, because we may not return to the starting state, it might be impossible to identify the correct starting state of the machine.

Known Limitations. Due to the nature of how the distinguishing of different states with the same output is performed in Algorithm 11, we require at least one reachable state with a *unique output*. We can use such uniquely identified states to then distinguish other states with duplicated outputs using the input sequences from their corresponding W-set, which is explained in more detail in Section 3.10.

For the usage of this algorithm with industrial measurement devices, such as the Smoke Meter in Section 6.2, this limitation has not impaired us yet as these devices have a central *ready* state, or equivalent thereof, which is ordinarily identifiable by its unique output.

It might be possible to remove this limitation in the future in a more refined version of the algorithm as addressed in Section 7.4.

3.2 Data Structures

The majority of the algorithms work is done on a number of different data structures, most of them comprised of sets, mappings and sequences. These data structures are mostly required to be used together because changes are done to a number of data structures at once. To simplify the passing of arguments to different functions we pack all the required data structures into one larger data structure we will call *context*. Passing *context* to any function as an argument will be equivalent to passing the individual data structures to it instead.

We define *context* as the following 9-tuple:

$$context = (I, O, S, \delta_n, \lambda, s_c, R, A, Tr) \tag{3.1}$$

where its individual parts are:

- I is the finite, non-empty set of input symbols
- O is the set of observed output symbols
- S the set of states
- $\delta_n : S \times I \to \mathcal{P}(S)$ is the non-deterministic state transition function mapping each state and input to a set of possible transition states
- $\lambda : S \to O$ is the output function
- $s_c \in S$ is the current state which the SUL is believed to occupy at the moment
- $R : O \to W$ where W is an *output specific characterizing set* (see Section 3.3.5) which can be used to distinguish between all states with the same output $S[o] = \{s \mid s \in S \wedge \lambda(s) = o\}$ for a given $o \in O$
- $A \subset S$ containing all abandoned states
- $Tr : (O \times I \times O)^*$ is the global *trace* which is the sequence of steps taken during the algorithm from a state with output $o^{src} \in O$ using an input $i \in I$ to a state with a given output $o^{dest} \in O$. $Tr = \langle (o_1^{src}, i_1, o_1^{dest}), (o_2^{src}, i_2, o_2^{dest}), ..., (o_n^{src}, i_n, o_n^{dest}) \rangle$ where each o_k^{src} is the output of a state from which the input i_k led to a state with output o_k^{dest}. Every step and their corresponding input and outputs are recorded in the trace such that the number of steps observed during the algorithm is n

3.3 Definitions

3.3.1 Abandoning States

Abandoning states is a mechanism to preserve states that are seemingly unreachable. Instead of simply deleting the states and potentially losing information that was already gathered about them, we abandon them by adding them to the set A. Abandoned states are for all intents and purposed ignored, whether in the definitions about the transition function or in the navigation of the SUL. They are only of interest if we find one of these abandoned states again at a later point. In that case we *recover* an abandoned state by removing it from the set A and using it again going forward.

The reasons why a state might be unreachable are that a state might not be part of the strongly-connected component we are currently learning, in which case the state will stay abandoned, or we simply do not know enough about the SUL to navigate back to a given state. We might at a later point rediscover the abandoned state, either stumbling about the state by chance or finding a path to it via an equivalence query.

3.3.2 Defining Determinism

We consider a state $s \in S$ to be *deterministic* if its transition function maps to at most one state for each input, i.e., there is no ambiguity to which state any transition leads from that state.

$$s \in S \text{ is deterministic} \iff \forall i \in I : |\delta_n(s,i)| \leq 1 \tag{3.2}$$

We can use this property to decide if the entire transition function is *deterministic*:

$$\delta_n \text{ is deterministic} \iff \forall s \in (S \setminus A) : s \text{ is deterministic} \tag{3.3}$$

We ignore abandoned states for such decisions as they are for all intents and purposes not part of the hypothesis at this moment as described in Section 3.3.1. Knowing whether the transition function is deterministic is important for deciding if a hypothesis can be extracted from *context*.

3.3.3 Defining Input Completeness

Similarly we can define if a state is *input-complete*:

$$s \in S \text{ is input-complete} \iff \forall i \in I, \exists s' \in \delta_n(s,i) : s' \notin A \tag{3.4}$$

A state is input-complete if there is at least one transition to a not abandoned state for every input. It is not enough to simply have any state in the transition function as only having abandoned states would indicate that we do not know where this transition leads to (as the target could not be reached at an earlier point.)

Again, we can define this property for the entire transition function while ignoring abandoned states:

$$\delta_n \text{ is input-complete} \iff \forall s \in (S \setminus A) : s \text{ is input-complete} \tag{3.5}$$

3.3.4 Constructing Deterministic Hypothesis

At any point, if δ_n is *deterministic* and *input-complete*, a Moore machine may be extracted from $context$ as a hypothesis which may then be evaluated by the equivalence oracle.

The hypothesis is the Moore machine $M = (I, O, S', s_0, \delta_k, \lambda)$ where I, O and λ can be directly taken from $context$. S' is simply the set $S \setminus A$ while δ_k is defined as:

$$\delta_k = \forall s \in (S \setminus A), \forall i \in I, \exists! k \in \delta_n(s,i) : \delta_k(s,i) = k \tag{3.6}$$

The initial state s_0 can freely be chosen from among the reachable states $s_0 \in (S \setminus A)$, although there are different strategies of selecting it. We could, for example, use information from the trace to restrict the choice of s_0 or alternatively to choose a state that is well-connected to other states in order to have the shortest possible reset sequence for a majority of states.

During each equivalence query it is important to navigate the SUL to the same initial state s_0 that was chosen for the hypothesis when the equivalence oracle requires a reset as described in Section 2.2.

3.3.5 Output Specific Characterization Sets

The classical W-set was already introduced in Section 2.5, which can distinguish any two states from each other based on their different responses to its input sequences. Distinguishing two states requires applying all input sequences of the W-set and observing the responses to them, so a small W-set is desirable, as a large W-set would mean performing many more queries for every state.

Therefore, instead of building one large W-set with sequences for all states, we maintain one *output specific characterizing set* or W_o-set for each unique observed output. A W_o-set contains input sequences called *distinguishers* $d \in I^+$ which can distinguish all states with the same output $S[o] = \{s \mid s \in S \wedge \lambda(s) = o\}$ for any $o \in O$. The W_o-sets are maintained in R of *context* as shown in Section 3.2.

The advantage that a mapping of W_o-sets, like R, has over a single classical W-set is that we only perform the distinguishers necessary to differentiate between states with the same output, which results in fewer queries during the distinguishing process. For example, if some output $o \in O$ was unique, i.e., we do not have distinguishers for it as $R(o) = \emptyset$, then any state with that output would not have to be distinguished as its output o makes it uniquely identifiable already. In contrast, if we were to use a classical W-set we would have to perform all distinguishers on every state no matter if its output was unique or not.

3.4 Main Algorithm

The main learning procedure is depicted in Algorithm 1. The high-level idea described by it can be summarized in the following steps:

1. Explore the SUL such that δ_n is *input-complete*
2. If δ_n is *deterministic* we query the equivalence oracle
3. Otherwise we calculate the differences between our observations and the hypothesis and refine it by splitting states with identical outputs as required

These steps can then be repeated until either the equivalence oracle does not find a counterexample to the provided hypothesis, in which case we return the result, or we fail to refine the hypothesis further, in which case we terminate the algorithm.

Algorithm 1 requires the following arguments:

- The input alphabet I
- The `sul` object which provides us with an interface to the actual system under learning by supplying the `init` and `step` methods
- An equivalence `oracle` which can find counterexamples, i.e., differences in behavior between the SUL and a provided hypothesis
- The parameter *len*$_{max}$ which decides the maximum length of words that we extract from the trace to navigate to certain states

SUL and Oracle Arguments. The `sul` argument provided as input is the interface to the system under learning in the form of the *output oracle* described in the MAT framework in Section 2.2. It provides the following two interactions:

- `init` which is an empty output query and returns the output of the current state without changing the state of the underlying SUL
- `step` which takes a single input $i \in I$, provides it to the underlying SUL and returns the output of the new state. In effect, it is an output query of size one without reset.

The `oracle` argument provided is a wrapper for an *equivalence oracle*. It provides the `findCex` method which wraps an implementation of any testing algorithm, e.g., the W-method by Chow [16], with the following changes:

Algorithm 1 Main Algorithm.

```
1: function MAINALGORITHM(I, sul, oracle, $len_{max}$)
2:     $o_0 \leftarrow$ sul.init()                               ▷ Provide initial output without changing state
3:     $context \leftarrow (O \leftarrow \{o_0\}, S \leftarrow \{s_0\}, \delta_n \leftarrow \emptyset, \lambda \leftarrow \{s_0 \mapsto o_0\},$
4:             $s_c \leftarrow s_0, R \leftarrow \{o_0 \mapsto \emptyset\}, A \leftarrow \emptyset, Tr \leftarrow \langle\rangle)$           ▷ Create $s_0$ with context
5:     changedTransitions, changedW, changedStates, changedLastRound $\leftarrow$ True
6:     while changedTransitions $\vee$ changedW $\vee$ changedStates $\vee$ changedLastRound do
7:         changedLastRound $\leftarrow$ changedTransitions $\vee$ changedW $\vee$ changedStates
8:         (changedTransitions, context) $\leftarrow$ exploreMissing(sul, context, $len_{max}$)    ▷ See Algorithm 2
9:         $A \leftarrow$ abandonUnreachableStates($I, S, \delta_n, s_c, A$)              ▷ See Algorithm 3
10:        if not deterministic $\delta_n$ then                              ▷ See Section 3.3.2
11:            (changedW, R) $\leftarrow$ calculateSpecificWsets(R, Tr)            ▷ See Algorithm 10
12:            (changedStates, context) $\leftarrow$ distinguish(sul, context, $len_{max}$)   ▷ See Algorithm 11
13:        else
14:            hypothesis $\leftarrow$ constructHypothesis(context)               ▷ See Section 3.3.4
15:            (cex, context) $\leftarrow$ oracle.findCex(sul, hypothesis, context)      ▷ See Section 3.5
16:            if $cex = \emptyset$ then
17:                return hypothesis
       return $\emptyset$                         ▷ No further changes detected. Possibly increase $len_{max}$
```

- Before the testing algorithm performs an output query, we perform a *reset sequence* using Algorithm 5 to find a transfer sequence to the chosen starting state s_0 of the hypothesis and perform Algorithm 7 to navigate to it.
- For each input of every output query that the equivalence oracle performs, we use Algorithm 9 `updateStep` to update our context to reflect the changes in the underlying SUL.
- If at any point during a reset sequence or an output query we observe unexpected behavior from Algorithm 9 we may stop the equivalence check as this means that we found a difference between the *context*, which is at that point equivalent to the constructed hypothesis, and the SUL.
- Finally, it returns the updated context.

Termination. The main algorithm terminates if the equivalence oracle cannot find a counterexample anymore in which case the constructed hypothesis will be returned. This is the successful termination of the algorithm. Alternatively, there are two failing cases in which the algorithm also terminates:

It is possible that the algorithm gets stuck in a state where distinguishing states is not successful, e.g., if navigation to a state is not possible with the given *len*$_{max}$, in which case no further changes would be made. This can be detected by the *changedTransitions*, *changedW* and *changedStates* variables in Line 7 which would all be false after one iteration of the loop and thus terminate with no hypothesis being returned. Most such cases can be prevented by choosing a large enough *len*$_{max}$.

The algorithm also terminates if during the distinguishing of states in Algorithm 11 in Line 7 the hypothesis does not have a reachable state with a unique output. As explained in Section 3.1 this is one of the known limitations of the algorithm.

3.5 Finding Counterexamples

In Algorithm 1 in Line 15 we query the equivalence oracle, which is a wrapper for the implementation of a testing algorithm as explained in Section 3.4, to find a counterexample to the provided deterministic hypothesis. Equivalence queries are implemented via a sequence of steps on the SUL and resets. To keep our context synchronized with the current state of the SUL we perform the `updateStep` method

(Algorithm 9) every time the equivalence oracle takes a step with the SUL. This allows us to keep track of the current state of the SUL, detect inconsistencies with the hypothesis immediately and allows us to navigate the SUL into the initial state of the hypothesis when a reset is required.

Whenever the equivalence oracle wants to reset the SUL we execute `navigateToTarget` (Algorithm 4) with the initial state of the hypothesis s_0 that we chose when constructing the hypothesis in Line 14. If at any point during the navigation to the initial state or during any of the steps performed by the equivalence oracle we detected *unexpected* behavior via the `updateStep` method (Algorithm 9) then we found a counterexample. Because we perform the `updateStep` method for every single step during the counterexample finding, thus updating the trace Tr with every step, we already processed any counterexample and incorporated it into the trace from which new distinguishing sequences can be computed later using Algorithm 10. This applies to normal counterexamples found during output queries but also for unexpected behavior found during the reset sequences.

One of the upsides of performing a reset sequence as described above is that we may use any testing algorithm for our equivalence oracle, even if it uses resets. In fact, the equivalence oracle does not even realize that anything is amiss, as the resets are simulated whenever they are required.

Counterexample Processing. The reason why there is no algorithm specifically to process a counterexample is because we detect unexpected behavior online due to the usage of Algorithm 9 during every step of the equivalence check. In fact, the counterexample is not really needed for our algorithm and we use only the absence of a counterexample to determine if we should return the finished hypothesis in Line 16.

Instead we use the mechanism of unexpected behavior, which is introduced in Section 3.8, to identify new states and transitions using Algorithm 9. The unexpected observation is integrated into the transition function δ_n and the trace Tr on every step, such that our model reflects the new outputs immediately. Finding unexpected behavior, i.e., making an observation that does not fit our current model of the SUL, during an equivalence query is not treated differently than finding it during exploration or navigation, in that the newly discovered state or transition must be explored and/or distinguished, i.e., until the hypothesis is deterministic and input-complete again. In short, we use the `findCex` method not find a counterexample directly, but for the testing strategy and the observations that we can make during the equivalence query.

We still use the `findCex` method, even though we do not need the counterexample, because most common equivalence oracles, such as the ones implemented by automata learning libraries AALpy [28] or LearnLib [22], provide this method by default. We wanted to allow the usage of any of these equivalence oracles in place, with only the changes described in Section 3.4, which can be done with a wrapper without modifying the actual testing algorithm.

3.6 State Exploration

The state exploration performed in Algorithm 2 can be described in the following steps:

1. While the transition function is not input-complete according to Equation 3.5
2. Navigate to a state that is not input-complete and perform a step to explore a yet unexplored input
3. Alternatively, abandon the target state if it could not be navigated to

The target states are chosen by performing a breadth first search on the known states, targeting the closest non input-complete state every time until no further targets can be found. States with the same output will be treated as the same state, thus seeming non-deterministic behavior can occur during the algorithm, e.g., multiple transitions on the same input from the same state to different targets. This may lead to the case were navigation to a target state fails. If a target cannot be reached, either because no path is known to that state or because of this seeming non-determinism, it is instead abandoned in Algorithm 4 in Line 11. Abandoned states are ignored as targets but may be found at a later point during navigation in which case they are *restored*.

Algorithm 2 State Exploration.

```
 1: function EXPLOREMISSING(sul, context, $len_{max}$)
 2:     changed ← False
 3:     repeat
 4:         targets ← $\{s \mid s \in (S \setminus A) \wedge \neg(s$ is input-complete$)\}$        ▷ As defined in Section 3.3.3
 5:         (word, $s^{target}$) ← BFS($s_c, targets, I, \delta_n, A$)                ▷ See Algorithm 5
 6:                        ▷ Find a word to a state with missing transitions or to only abandoned
 7:         if ($s^{target} \neq$ None) then
 8:             context ← navigateToTarget($s^{target}$, sul, context, $len_{max}$)         ▷ See Algorithm 4
 9:             changed ← True
10:             if $s_c = s^{target}$ then
11:                 choose $i \in I : (\delta_n(s_c, i) = \emptyset \vee \forall s' \in \delta_n(s_c, i) : s' \in A)$
12:                 (expected, context) ← takeStep($i$, sul, context)              ▷ See Algorithm 8
13:     until ($s^{target}$ = None)
14:     return (changed, context)
```

Termination. Let M be the underlying Moore machine of the SUL which has n states in total. In each iteration of the loop in Line 3 we try finding a path to one of the states that are not input-complete yet. Let the size of that set be $t = |targets|$ in Line 4. The set of states that we can actually find a path to in our graph using Algorithm 5 is potentially smaller if the graph is not strongly-connected, or if we have not found any transitions to some states yet. Let this set be $reachable \subseteq targets$ with $r = |reachable|$ and $r \leq t$. Finally, let m be the number of transitions which still have to be explored in order to make δ_n input-complete as described in Equation 3.5. If $m = 0$ the loop terminates and δ_n will be input-complete.

Firstly, m is bounded by $t * |I|$ and t is in turn bounded by n. If we discover new outputs and generate new states with Algorithm 9 then m and t may temporarily grow but only until all outputs have been discovered. Additionally, the number of new transitions that can be discovered is bounded by $n * |I|$ which means that unknown transitions cannot be discovered without end.

Secondly, an important insight is that reaching any state $s \in reachable$, which will in turn lead to trying a new input on s in Line 11 and 12, will always bring s closer to input-completeness as m will decrease by 1.

Discovering new outputs and reaching states to try their inputs will eventually lead to an $m = 0$. The special cases that have to be taken into account occur when a state was not reached via Algorithm 4 in Line 8. If a state cannot be reached it is *abandoned* in Line 11 and no longer counts towards input-completeness of δ_n thus decreasing t by 1.

Abandoning a state s_a will change m in two ways:

1. m will increase by the number of transitions of other states that had s_a as target and have no other reachable states on that input
2. m will decrease by the number of transitions that were still missing on s_a

Restoring a state s_a will do the opposite, namely:

1. m will decrease by the number of transitions of other states that have s_a as target which now count towards input-completeness again
2. m will increase by the number of transitions that we now have to explore on the newly restored state s_a

Abandoning s_a may lead to an increase in m as described in Case 1. If s_a is not restored again then we simply have one less target to explore. We may have to visit some other states again but just abandoning states will eventually lead to input-completeness, as at most $t - 1$ states can be abandoned at the same time.

Otherwise, to restore an abandoned state we have to actually visit that state as can be seen in Algorithm 9 in Line 5, which is the only place were states are restored. If we do restore a state via Case 2 and still have missing inputs in said state we may now try one of them and decrease m by 1. Therefore, abandoning and restoring the same state multiple times will at some point leave that state input-complete while recovering a state that is already input-complete will never increase m. Only non-input complete states may be targeted, and only targeted states may be abandoned, which prevents endlessly looping between abandoning a state and restoring it again.

Thus at some point abandoning a targeted state s_a, which must have missing transitions as it would not have been targeted otherwise, will no longer increase m via Case 1 but only decrease it via Case 2. Similarly, restoring an input-complete state s_a, and it will be input-complete after restoring it often enough, will not increase m via Case 2 but only decrease it via Case 2, guaranteeing the termination of Algorithm 2.

Abandoning Remaining States. Once Algorithm 2 terminates some states may be unreachable which were not abandoned yet. Specifically, if Algorithm 2 terminates by abandoning a state, i.e., a state is abandoned which makes the transition function input-complete, then all states that are only reachable through this abandoned state are not reachable anymore and should thus be abandoned as well. This special case is dealt with by Algorthm 3 which is performed in the main algorithm after the execution of Algorithm 2 in Line 9.

Note that performing Algorithm 3 while the transition function is input-complete, as described in Equation 3.5, will not change this property of the transition function as only states which cannot be reached will be abandoned.

Algorithm 3 Abandon Unreachable States.

```
1: function ABANDONUNREACHABLESTATES($I, S, \delta_n, s_c, A$)
2:     for all $s \in (S \setminus A)$ do                    ▷ For all non-abandoned states
3:         ($word$, $target$) ← BFS($s_c, \{s\}, I, \delta_n, A$)    ▷ Breadth first search, see Algorithm 5
4:         if $target$ = None then                           ▷ No path exists from $s_c$ to $s$
5:             $A \leftarrow A \cup \{s\}$                   ▷ Abandon unreachable state
    return $A$
```

3.7 Navigating the SUL

In Algorithm 4 we show the main procedure of reaching a state s^{dest} from our current state s_c. If at any point during the navigation we encounter unexpected behavior, either finding a new transition to a state with a new output or to an existing state, or by recovering an abandoned state as shown in Algorithm 9, we stop the navigation and return. This is because we want to take special action, especially during Algorithm 11 where we stop and return to the main algorithm, when we encounter unexpected behavior. There are two different methods of how navigation to a state is performed:

The first method, and the one always tried initially, is to directly navigate to the target using the shortest path calculated in Line 7. We search through the transition function using breadth first search with Algorithm 5 and try navigating the resulting sequence of inputs step by step.

However, due to δ_n potentially being non-deterministic, there is no guarantee that the shortest path actually leads us to our target state, i.e., there might be multiple transitions on any given state and input. In this case, we may try an alternative navigation method using words from the trace instead which are

Algorithm 4 Navigate to target, first trying the direct path then trying different paths recorded in the trace. Stop if the target is reached, unexpected behavior occurs or all possible words for a state have been tried.

```
1: function NAVIGATETOTARGET($s^{dest}$, sul, context, $len_{max}$)
2:     $P : S \to words$    ▷ Contains all words from trace for a given $s \in S$ to all states with output of $s^{dest}$ with a maximum length of $len_{max}$
3:     $expected \leftarrow$ True
4:     while $s_c \neq s^{dest} \wedge expected$ do
5:         if $P(s_c) =$ undefined then
6:             $P(s_c) \leftarrow$ findAllWords($s_c, s^{dest}, \lambda, Tr, len_{max}$)    ▷ See Algorithm 6
7:             ($directWord$, $s^{target}$) $\leftarrow$ BFS($s^{source}$, $\{s^{dest}\}, I, \delta_n, A$)    ▷ See Algorithm 5
8:             if $s^{target} = s^{dest}$ then    ▷ Found a path
9:                 ($expected$, _, $context$) $\leftarrow$ navigateWord($directWord$, $s^{dest}$, $sul$, $context$)
10:        else if $|P(s_c)| = 0$ then
11:            $A \leftarrow A \cup \{s^{dest}\}$    ▷ All words tried but could not be reached, so abandon
12:            return $context$
13:        else
14:            choose $word$ : $word \in P(s_c)$
15:            $P(s_c) \leftarrow P(s_c) \setminus \{word\}$
16:            ($expected$, _, $context$) $\leftarrow$ navigateWord($word$, $s^{dest}$, $sul$, $context$)    ▷ See Algorithm 7
17:            if $\neg expected$ then return $context$
18:    return $context$
```

calculated in Line 6. Hereby we find words that resulted in sequences starting with the output of our current state and ending in the output of the target state with a maximum length of len_{max}. One of these sequences should lead us back to the target state as we observed these sequences of inputs and outputs before.

If both navigation methods do not lead us back to the target state, we abandon it instead. This allows us to ignore it for now. It is possible to recover it at a later point either by chance or during an equivalence query. Reasons why a state cannot not be reached could be that the target is not part of a strongly-connected component, in which case the state will stay abandoned, or we did not observe a sequence yet which leads us to the state or the sequence that does lead us there is longer than len_{max} and was thus not tried.

Finding Direct Word. Algorithm 5 performs breadth first search on the non-deterministic transition function δ_n and returns the shortest word from s^{src} to any of the states in S_{dest} as well as to which target the word leads. Abandoned states are ignored while every other reachable state is only visited once. If none of the target states are reachable via normal transitions then an empty sequence and **None** are returned instead.

Due to the nature of the non-deterministic transition function δ_n there is no guarantee that the shortest word will actually lead to the target state as there could potentially be multiple possible transitions on the same input. Therefore, Algorithm 4 uses the direct word first and then tries an alternative method of extracting words from the trace as explained in Section 3.7 if the direct word does not successfully reach the target.

Finding Words from Trace. Algorithm 6 searches through the trace to find sequences that have led to states with the output of the target state from a state with the current output in the past. This alternative method of finding a word, i.e., transfer sequence, that leads to a target state is sometimes necessary if the direct word to the target state does not work, potentially because of seeming non-determinism, i.e.,

Algorithm 5 Breadth first search for a word from s^{src} to any state in set S_{dest} ignoring transitions over abandoned states.

```
1: function BFS($s^{src}, S_{dest}, I, \delta_n, A$)
2:     $V \leftarrow \emptyset$                                        ▷ Set of visited states
3:     $q_s \leftarrow \langle s^{src} \rangle$                         ▷ Queue of states to check next
4:     $q_w \leftarrow \langle \langle\rangle \rangle$                  ▷ Queue of built words
5:     while $|q_s| > 0$ do
6:         $s_{next}, q_s \leftarrow$ dequeue($q_s$)     ▷ Return and remove from $q_s$ the first element
7:         $word, q_w \leftarrow$ dequeue($q_w$)
8:         if $s_{next} \in S_{dest}$ then return ($word, s_{next}$)
9:         $V \leftarrow V \cup \{s_{next}\}$
10:        for all $i \in I$ do
11:            for all $s \in \{s \mid s \in \delta_n(s_{next}, i) \wedge s \notin A \wedge s \notin V\}$ do
12:                $q_s \leftarrow q_s + \langle s \rangle$                    ▷ Append $s$ to $q_s$
13:                $q_w \leftarrow q_w + \langle word + \langle i \rangle \rangle$   ▷ Append word extended by $i$ to $q_w$
       return ($\emptyset$, None)                    ▷ No path found from $s^{src}$ to any $S_{dest}$
```

multiple transitions on the same input in the transition function δ_n.

The maximum length of these words is restricted to *len*$_{max}$, which is given as an argument to Algorithm 1. An example where just the direct word to the target, i.e., *len*$_{max} = 0$, is not sufficient to learn a Moore machine correctly is shown in 4.2.

Algorithm 6 Find all words in trace leading from observed output $\lambda(s_c)$ to $\lambda(s^{dest})$ with a maximum length of *len*$_{max}$.

```
1: function FINDALLWORDS($s_c, s^{dest}, \lambda, Tr, len_{max}$)
2:     $words \leftarrow \emptyset$
3:     for $k = |Tr|$ to 2 do
4:         $(o_k^{src}, i_k, o_k^{dest}) \leftarrow Tr(k)$
5:         if $o_k^{dest} = \lambda(s^{dest})$ then
6:             $word \leftarrow \langle\rangle$
7:             for $l = 0$ to min($len_{max}, k - 1$) do
8:                 $(o_l^{src}, i_l, o_l^{dest}) \leftarrow Tr(l)$
9:                 $word \leftarrow \langle i_l \rangle + word$
10:                if $o_l^{str} = \lambda(s_c)$ then
11:                    $words \leftarrow words \cup \{word\}$
       return $words$
```

Navigate Word. Algorithm 7 applies an input sequence *word* to the SUL step-wise, aborting the procedure if unexpected behavior occurs or if the target state could not be reached, and returns its response. Every step is taken via Algorithm 8, which in turn updates *context* and keeps track of the current state. The procedure is similar to a step-wise performed output query, with the exception that the navigation can be aborted before the entire input sequence was applied.

Applying an Input. Algorithm 8 applies a single input to the SUL, updates the *context* appropriately in Line 3 and returns if the observed output is consistent with our existing hypothesis. Note that *sul*.step is equivalent to an output query of length 1 without reset as described in Section 3.4, in that it applies the provided input to the SUL and returns the output of the newly reached state. Algorithm 8 is the only procedure that performs *sul*.step directly, all other algorithms use this one instead and so automatically update *context* with every step.

Algorithm 7 Navigate a $word \in I^*$ stopping if any unexpected behavior occurred and returning if the state s^{dest} was reached and the response sequence of observed outputs.

```
1: function NAVIGATEWORD(word, $s^{dest}$, sul, context)
2:     $response \leftarrow \langle\rangle$
3:     for $k = 1$ to $|word|$ do
4:         $i \leftarrow word(k)$                                        ▷ $k^{th}$ element of sequence word
5:         ($expected$, $context$) $\leftarrow$ takeStep($i$, $sul$, $context$)        ▷ See Algorithm 8
6:         $response \leftarrow response + \langle\lambda(s_c)\rangle$
7:         if not $expected$ then return (False, $response$, $context$)
8:     return ($s_c = s^{dest}$, $response$, $context$)
```

Algorithm 8 Take a single step with the `sul` with $i \in I$, update context appropriately and return if the transition was expected, i.e., a state with the correct output was already in the transition function of the current state.

```
1: function TAKESTEP($i$, sul, context)
2:     $o_c \leftarrow sul$.step($i$)                                ▷ Take step, observe output
3:     ($expected$, $context$) $\leftarrow$ updateStep($i$, $o_c$, $context$)      ▷ See Algorithm 9
4:     return ($expected$, $context$)
```

3.8 Updating Context

Algorithm 9 takes the input and corresponding output of a single step on the SUL and updates *context* appropriately. It is one of the central procedures and will not only update the current state but add new transitions, states, outputs as required in addition to recording steps in the trace Tr. Not only is this procedure performed after every step during the learning process in Algorithm 8 but also while equivalence testing as explained in Section 3.5. In short, every step taken on the SUL must have a corresponding update of *context* via Algorithm 9.

An important part of the algorithm is that it chooses the new current state s_c. It is guaranteed that after its execution s_c has the newly observed output $o_c = \lambda(s_c)$ even though the state itself might not be the correct one as it could have been chosen in Line 9. It is then the task of Algorithm 11 to actually distinguish the states with the same outputs. Note that if we visit an abandoned state then it will immediately be restored in Line 5. We only abandon states because we cannot reach them at some point in time and reaching it later might let us reach it in the future as well.

Expected and Unexpected Behavior. There are two different cases, namely expected and unexpected behavior, for observing a new step and updating *context* using Algorithm 9. Expected behavior means that a step was consistent with the existing hypothesis while unexpected behavior refers to a new discovery, like a new transition or state, or the restoration of a previously abandoned state.

Generally, if there already is a state in the transition function of the current state with the given input which leads to a state that has the observed output then the observation does not change anything for our hypothesis, i.e., it was *expected*. More formally, a step is *expected* iff

$$\exists s \in \delta_n(s_c, i) : \lambda(s) = o \wedge s \notin A \tag{3.7}$$

where s_c is the current state, $i \in I$ is the input provided to the SUL and $o \in O$ the corresponding output of the new state. In this case, only the current state will be updated in Line 4 and an entry added to the trace in Line 18.

Otherwise, if the observation is not consistent with our current hypothesis then we have to change it in order to match the observation. The possible observations that require changes to *context*, and are therefore *unexpected*, are:

Algorithm 9 Update the context to reflect a single step taken with $i \in I$ and the newly observed output $o_c \in O$ and return if the transition was expected, i.e., a state with the correct output was already in the transition function of the current state.

```
1: function UPDATESTEP($i$, $o_c$, context)
2:     $s^{last} \leftarrow s_c$                              ▷ Last state before taking step is $s_c$
3:     if $\exists s \in \delta_n(s^{last}, i) : \lambda(s) = o_c$ then
4:         $s_c \leftarrow s$
5:         $expected \leftarrow s_c \notin A$
6:     else                  ▷ No transition from $s^{last}$ to state with output $o_c$ using $i$ exists
7:         $expected \leftarrow$ False                        ▷ A new transition not expected
8:         if $o_c \in O$ then
9:             $s_c \leftarrow S[o_c]$.first          ▷ Choose the first state with output $o_c$, see Case 3
10:        else                                          ▷ $o_c$ was never observed before
11:            $O \leftarrow O \cup \{o_c\}$                         ▷ Extend output alphabet
12:            $s_c \leftarrow s^{new} : s^{new} \notin S$                  ▷ Create a new state $s^{new}$
13:            $S \leftarrow S \cup \{s_c\}$
14:            $R \leftarrow R \cup \{o_c \mapsto \emptyset\}$
15:            $\lambda \leftarrow \lambda \cup \{s_c \mapsto o_c\}$
16:        $\delta_n \leftarrow \delta_n \cup \{(s^{last}, i) \mapsto s_c\}$          ▷ Add $s_c$ to the set of target states
17:    $A \leftarrow A \setminus \{s_c\}$                               ▷ Recover state if abandoned
18:    $Tr \leftarrow Tr + (\lambda(s^{last}), i, o_c)$                       ▷ Add new step to trace
19:    return expected, context
```

1. Restoring a previously abandoned state in the transition function of the current state in Line 5
2. Observing a never before seen output after Line 10
3. Observing an output that is not in the transition function of the current state in Line 8

All of these cases require that our current model of the SUL, represented by *context*, must be changed in order to reflect the new observation:

For Case 1, we simply restore an abandoned state again if it is in the transition function of the current state. We prefer restoring an abandoned state to adding an additional transition to a state with the same output as we do not want to have multiple states with the same output in a single transition. More formally, we require that the following property holds

$$\forall s \in S, \forall i \in I, \forall o \in O : |\{s' \mid s' \in \delta_n(s, i) \land \lambda(s') = o\}| \leq 1 \tag{3.8}$$

By trying this case first we never have the situation that there could be multiple states with the same output in Line 3.

Regarding Case 2, if we observe a new output, that has never been seen before, then we have to construct a new state with said output and add it to the transition function of the current state, extending the output alphabet O in the process. The existence of a new state must also be reflected in S, λ, δ_n and R. Observing a new output will happen especially often in the first iteration of Algorithm 2 where outputs are discovered frequently.

Finally, in Case 3, if we observe an output which is not in the transition function of the current state, i.e., we have not observed this output to the given input from this state, then we create a new transition to an existing state with the observed output. Generally, the algorithm assumes, if possible, that an existing state was instead of a new state being found. Therefore, if there is at least one state with the observed output already, the transition is added to the existing state. The exact state that should be chosen from the set of states with the same output $S[o]$ in Line 9 with the observed output $o \in O$ should be the same across different executions of Algorithm 9.

This is important for Algorithm 11 as during the exploration of transitions of a newly split state in Line 17 we require that new transitions are made to the states that were created during the merge in Line 5. These are the states that we gradually split up and not forming transitions to these states would leave newly found ones incorrectly distinguished.

There are two methods of choosing these states out of the set $S[o]$:

The chosen state $s^{first} \in S[o]$ should be the *earliest created state* from the states in $S[o]$. If every state has a unique identifier that corresponds to the order in which they were created then the state with the lowest identifier should be chosen first. This has the intended result because all states split in Algorithm 11 in Line 16 will have been created after the states in Line 5 and so will not be chosen as target states.

Alternately, $s^{first} \in S[o]$ may be chosen to have the highest *transition number* which is defined for any state $s \in S$ as $\sum_{i \in I} |\delta_n(s,i)|$. Using the state with the highest transition number as the target for new transitions which also corresponds to the merged states in Line 5 which have the most transitions after the merge.

3.9 Calculating Output Specific W-sets from Trace

Algorithm 10 Calculate output specific W-sets according to trace and extend R.

```
 1: function CALCULATESPECIFICWSETS(R, Tr)
 2:     $R' \leftarrow R$
 3:     $C \leftarrow \emptyset$        ▷ Set of pairs of indices for which trace entries match in $o^{src}$ and $i$ but not in $o^{dest}$
 4:     for $k = 1$ to $|Tr|$ do
 5:         for $l = k + 1$ to $|Tr|$ do
 6:             $(o_k^{src}, i_k, o_k^{dest}), (o_l^{src}, i_l, o_l^{dest}) \leftarrow Tr(k), Tr(l)$
 7:             if $o_k^{src} = o_l^{src} \wedge i_k = i_l \wedge o_k^{dest} \neq o_l^{dest}$ then
 8:                 $C \leftarrow C \cup \{(k,l)\}$
 9:     for all $(k,l) \in C$ do
10:         $d \leftarrow \langle\rangle$        ▷ Distinguisher $d$ initially empty
11:         $(o_k^{src}, i_k, o_k^{dest}), (o_l^{src}, i_l, o_l^{dest}) \leftarrow Tr(k), Tr(l)$
12:         while $k >= 1 \wedge l >= 1 \wedge o_k^{src} = o_l^{src} \wedge i_k = i_l$ do
13:             $d \leftarrow \langle i_k \rangle + d$        ▷ Extend d by input $i_k = i_l$
14:             $R' \leftarrow R' \cup \{o_k^{src} \mapsto d\}$        ▷ Add d to $W_o$-set of output $o_k^{src}$
15:             $k, l \leftarrow k - 1, l - 1$        ▷ Now check predecessor
16:             $(o_k^{src}, i_k, o_k^{dest}), (o_l^{src}, i_l, o_l^{dest}) \leftarrow Tr(k), Tr(l)$
        return $(R \neq R', R')$        ▷ Return if $R'$ changed compared to $R$
```

Algorithm 10 calculates the output specific W-sets, W_o-sets, as explained in Section 3.3.5, from observations of the trace Tr and updates R, the mapping of each output to such a corresponding W_o-set, appropriately.

It does so by initially calculating C which is the set of all entries of the trace that show visible differences in their target state output o^{dest} but are identical in their source state output o^{src} and transition input i. Observing such a difference means that the source states of the two observed trace entries *cannot* be the same as our underlying Moore machine is deterministic which requires that there must exists multiple states that have the same output o^{src}. Furthermore, these two states can in the future be distinguished by applying the input i as it resulted in a visible difference in output o^{dest}. This distinguisher i is added to the W_o-set of o^{src} in the first iteration of the loop in Line 12.

Intuitively, if we observed the trace entries $t_1 = (o^{src}, i, o_1^{dest})$ and $t_2 = (o^{src}, i, o_2^{dest})$ with $o_1^{dest} \neq o_2^{dest}$ then we can conclude that there must be at least two states with the output o^{src}, as the same deterministic state could not have two different transitions on the same input. Therefore, the source states

Table 3.1: Example trace repeating states with output a until finally reaching b.

Index	o^{src}	i	o^{dest}
1	a	x	a
2	a	x	a
...	a	x	a
t	a	x	b

Table 3.2: Example trace alternating output a and others.

Index	o^{src}	i	o^{dest}
1	a	x	b
2	b	x	a
3	a	x	c
4	c	x	a
odd	a	x	...
even	...	x	a

in t_1 and t_2 cannot be the same, even though they have the output o^{src}, and that we may distinguish these two states in the future by applying the distinguisher i to them.

Additionally, we can also calculate distinguishers for the outputs in preceding entries of the pairs in C if they are identical. Similar to how having a visible difference between two trace entries t_1 and t_2 lets us distinguish the source states, having observed an identical sequence $\alpha \in Tr^+$ preceding the entries lets us do the same for all outputs in α. More formally, if there are two sequences αt_1 and αt_2 in the trace, then every output o^{src} for any entry w in α, where $\alpha = vwz$ with $v, z \in Tr^*$ and $w \in Tr$, can be distinguished by the sequence of inputs from wzt_1.

This can be seen in the example trace in Table 3.1 where the output a was observed t times, counting the starting state, followed by b, always applying the same input x. It can easily be reasoned that none of the states with output a can be the same and that the underlying Moore machine must have at least t states with the output a as any machine with fewer than t states with output a could not produce a sequence that changes only after t identical inputs. We can in fact distinguish every state s_k with output $o^{src} = a$ corresponding to Index k in that trace where $1 \leq k \leq t$ with a distinguisher $\langle x, ..., x \rangle$ of length $t - k$. In this special case, where every distinguisher shorter than $t - 1$ is a prefix of the distinguisher of length $t - 1$, we can even skip all but the longest distinguisher as explained in Section 3.9.

Runtime. We can easily define an upper bound for the runtime of Algorithm 10. Let $t = |Tr|$ and $c = |C|$. In Line 4 and 5 where both the outer and inner loop iterate through the entire trace thus performing $t * (t - 1)$ operations in total. With this loop our best case scenario for the runtime without any of the optimizations discussed in Section 3.9 is already up to $\mathcal{O}(t^2)$.

These almost t^2 loops could theoretically[1] result in t^2 entries in C if every pair of entries in the trace starts with the same output but ends in a different one thus resulting in $c = t^2$. Then performing c iterations in Line 9 with an inner loop in Line 12 which may in the worst case generate a distinguishing sequence d of length $t - 1$ if the second to last and last entry in the trace have to run backwards through the entire trace. Thus we can liberally define the upper bound at $\mathcal{O}(t^3)$.

In practice the runtime will be much lower, being closer to $\mathcal{O}(t^2)$. This is the case because there is a relation between c and how many iterations the loop in Line 12 can run. Generally, the larger c the fewer iterations Line 12 will perform and vice versa.

Table 3.1 and 3.2 contain two example traces Tr of length t that should illustrate this relation.

Table 3.1 has $c = t$ as every entry up to t is added in combination with entry t. However, it has the largest number of iterations possible in Line 12 namely $t - 1$ up to 1. So the runtime for the trace in Table 3.1 would be $t * (t - 1)$ for calculating C and $\sum_{k=1}^{t}(t - k) = \frac{1}{2}(t - 1)t$ for a total runtime of $\mathcal{O}(t^2)$.

In contrast, Table 3.2 has a much larger c but the maximum number of iterations in Line 12 are 1 as o^{dest} differs every other entry. This leaves the calculation of C as the primary factor of runtime which

[1]Actually, at most half of the entries can start with the same output and lead to a different one as at least one step is required to get back to a state with the original output, see Table 3.2

also results in $\mathcal{O}(t^2)$.

Optimizations. The runtime of the algorithm can be decreased drastically with these optimizations which can all be implemented at the same time. The first two reducing runtime of Algorithm 10 and the last one for Algorithm 11 instead.

- Introduction of *buckets* for the building of C which separate all entries in Tr into buckets with identical o^{src} and i such that each trace entry must only be compared within a single bucket instead of comparing it with all entries of the entire trace. Constructing the buckets takes linear time as a single iteration over the trace can separate out all entries into their corresponding buckets. This optimization works especially well if there are very few entries with the same o^{src} and i, so it would, for example, not improve the runtime of Table 3.1.
- As distinguishers are only ever added to R and never removed, it is unnecessary to re-compare old parts of the trace with each other as this will result in the same distinguishers again. Instead maintain a variable t_{last} that tracks the length of the trace at the moment when Algorithm 10 was last called. Then when creating C every entry of the trace must only be compared to the newly added parts, i.e., entries with their index greater than t_{last}. Additionally, this optimization is possible in conjunction with the bucket strategy. Buckets should only be created for entries of the trace with their index greater than t_{last} followed by comparing every entry of the trace to its corresponding bucket. This optimization does reduce the runtime for every execution of Algorithm 10 except the first one.
- A new distinguisher d should only be added to the corresponding W_o-set if it is not a prefix of any existing $w \in W_o$. Similarly, adding a new d to a W_o allows for the removal of all w for which w is a prefix of d. Intuitively, if w distinguishes between some states and d is a prefix of w: $|d| < |w|$ and $w = d\alpha$, then w must *at least* distinguish everything that d does, if not more, as any response sequence to d will be a prefix to the response sequence of w. While this does not reduce the runtime for Algorithm 10 itself, producing fewer distinguishers, i.e., smaller W_o-sets, will speed up Algorithm 11 instead.

3.10 Distinguishing States

Algorithm 11 takes the W_o-sets calculated in Algorithm 10 and applies their distinguishers to the corresponding states, splitting states whenever a new distinguisher is applied and finally merging states with the same responses to their applied W_o-set. If at any point during the navigation new unexpected behavior occurs, except during the exploration in Line 17, then the algorithm is stopped and returns to the main algorithm for another round of exploration and calculation of distinguishers. The algorithm works on *context* but not before creating a copy of it in *context'* in order to restore the original if the execution is aborted. This allows for liberal changes, adding and deleting states, because we retain a copy and may roll back to the original if we encounter difficulties such as unexpected behavior. To make sure that the copy *context'* reflects the operations performed on the underlying SUL, Algorithm 9 will be executed on *context'* whenever it is used on *context* using the same input and observed output as arguments. This way, the trace and other data structures are updated in the copy as well. If the algorithm is aborted at any point during the execution, state exploration can be performed on newly found outputs or new distinguishers can potentially be extracted from the now longer trace.

A state s is fully distinguished iff

$$|R(\lambda(s))| = |D(s)| \tag{3.9}$$

which means that all distinguishers in the W_o-set of $R(\lambda(s))$ were tried and their responses recorded in $D(s)$. Note that while the W_o-sets are identical across states with the same output every state may have a different number of observed responses in D.

Algorithm 11 Distinguish and split states according to R.

```
1:  function DISTINGUISH(sul, context)
2:      context'← context                                                    ▷ Make a copy of context
3:      whenever updateStep(..., context) → updateStep(..., context')        ▷ Whenever Algorithm 9 is
    performed on context it will also be performed on context' with the same arguments otherwise
4:      $D : S \to (W \to O^*) \leftarrow \emptyset$          ▷ D returns a partial mapping that is able to distinguish between
    states with equivalent output symbols if its domain $W = R(\lambda(s))$ with $s \in S$. Each input sequence
    $w \in W$ maps to an observed response sequence.
5:      $(S, \delta_n, \lambda) \leftarrow$ mergeStatesWithEquivalentOutput($O, S, \delta_n, \lambda$)      ▷ See Section 3.10
6:      while $\exists s \in S : |R(\lambda(s))| \neq |D(s)|$ do
7:          assert $\exists s \in (S \setminus A) : |R(\lambda(s))| = 0$                  ▷ See Section 3.1
8:          choose $s, i, s' : s, s' \notin A \land |R(\lambda(s))| = 0 \land s' \in \delta_n(s, i) \land |R(\lambda(s'))| > |D(s')|$
9:          context ← navigateToTarget($s$, sul, context)                       ▷ See Algorithm 4
10:         if $s_c \neq s$ then return (False, context')
11:         (expected, context) ← takeStep($i$, sul, context)    ▷ Enter state which we want to distinguish
12:         if $\neg expected \lor s_c \neq s'$ then return (False, context')
13:         choose $d : d \in R(\lambda(s')) \land d \notin D(s')$
14:         (expected, response, context) ← navigateWord($d$, None, sul, context)    ▷ See Algorithm 7
15:         if $\neg expected$ then return (False, context')
16:         $(s^{split}, D, S, \delta_n, \lambda) \leftarrow$ splitState($s'$, $d$, response, $s, i, D, S, \delta_n, \lambda$)    ▷ See Algorithm 12
17:         (_, context) ← exploreMissing(sul, context)                         ▷ See Algorithm 2
18:         $(D, S, \delta_n, \lambda) \leftarrow$ mergeIdenticalDistinguishedStates($s^{split}, D, S, \delta_n, \lambda$)    ▷ See Section 3.10
19:         context ← navigateToTarget($s$, sul, context)                       ▷ See Algorithm 4
20:         if $s_c \neq s$ then return (False, context')
21:         $(D, S, \delta_n, \lambda) \leftarrow$ removeUnreachableStates($D, S, \delta_n, \lambda, s_c, A$)    ▷ See Section 3.10
22:     return (True, context)
```

Algorithm 11 does the following until δ_n is deterministic or until unexpected behavior occurs:

1. Initially, copy *context* and merge all states with the same outputs in Lines 2 and 5.
2. Navigate to a state s' that is not fully distinguished, i.e., for which any distinguisher from its corresponding W_o-set was not yet tried, and enter it from a fully distinguished state s in Lines 8 to 12.
3. Apply said distinguisher to s', split off a new state s^{split} and add the applied distinguisher and its response to $D(s^{split})$. This new state is then one step closer to being distinguished as one more input sequence was observed for s^{split} than for s' from Lines 13 to 16.
4. Make δ_n input-complete again by exploring the transitions of the newly added state and merge it if it has identical transition outputs and distinguishers as an already existing state in Lines 17 and 18.
5. Finally, remove s' once it was entered via every available transition, eventually leaving only fully distinguished states in Line 21.

The algorithm terminates once all states with an non-empty W_o-set have been fully distinguished, which leaves δ_n deterministic again or unexpected behavior occurred, in which case new distinguishers can be calculated using Algorithm 10. A short example is presented later in this Section to better visualize how Algorithm 11 works.

Distinguishing States. To distinguish states we require W_o-sets, which are calculated in Algorithm 10. When a distinguisher d is applied to a state with output $o \in O$ it may then be differentiated from others in $S[o]$ comparing the response to d between them. To keep track of all responses obtained from applying distinguishers to states, D is introduced. D is a partial mapping that contains the responses for each distinguisher of every state as shown in Line 4. Once the response to every distinguisher $R(\lambda(s))$ was observed for $s \in S$ and recorded in D, as described in Definition 3.9, s is fully distinguished. The aim of Algorithm 11 is to fully distinguish all reachable states and thus return a deterministic δ_n.

Splitting States. Algorithm 12 is used to split states in Line 16. The split state s^{split} has the same output and distinguishers as the original except that additionally d was applied to it. We may add the response to d into $D(s^{split})$ which means that s^{split} is more distinguished than the state it was split off from. Because we entered s^{split} from a fully distinguished state s with input i we may now redirect the transition, which originally led to s', to s^{split} instead. This can be seen in the example in Section 3.10 in Figure 3.2 where State 0 originally lead to State 1 with input x but was then changed to lead to the newly split State 2 instead.

Algorithm 12 Split a state, copy its distinguishers and responses in D except for d and change the transition taken in Line 11 of Algorithm 11 to the newly split state.

```
1: function SPLITSTATE($s'$, $d$, response, $s$, $i$, $D$, $S$, $\delta_n$, $\lambda$)
2:     $s^{split} \leftarrow s^{new} : s^{new} \notin S$                          ▷ Create a copy of $s'$
3:     $S \leftarrow S \cup \{s^{split}\}$
4:     $\lambda \leftarrow \lambda \cup \{s^{split} \mapsto \lambda(s')\}$                    ▷ With the same output as $s'$
5:     $\delta_n \leftarrow \delta_n \cup \{(s^{split}, d(1)) \mapsto response(1)\}$ ▷ With only the first observation from $d$ in transition
6:     $D \leftarrow D \cup \{s^{split} \mapsto \{D(s') \cup \{d \mapsto response\}\}\}$   ▷ And add $d$ and its response to $D$
7:     $\delta_n \leftarrow \delta_n \setminus \{(s, i) \mapsto s'\}$                       ▷ Remove old transition from $s$
8:     $\delta_n \leftarrow \delta_n \cup \{(s, i) \mapsto s^{split}\}$           ▷ And replace with transition to new state $s^{split}$
9:     return ($s^{split}$, $D$, $S$, $\delta_n$, $\lambda$)
```

It is important that s is already fully distinguished, which leaves us with the limitation in Line 7 of Algorithm 11 as mentioned in Section 3.1. Otherwise, if s was not fully distinguished, we could not be sure if we identified s correctly and thus if we changed the the appropriate transitions in Lines 7 and 8. This would break Algorithm 11 as it stands at the moment, although it might be possible to remove this limitation in the future (see Section 7.4).

Merging States. Two states $s, s' \in S$ may only be merged if they have the same output, i.e., $\lambda(s) = \lambda(s')$. One of the states is chosen arbitrarily to be merged into the other state. For example, we may merge s' into s by adding all outgoing transitions from s' to s and by replacing all incoming transitions to s' with transitions to s instead. If s' is also the current state then it should be change to s instead. In other words, performing the following procedure:

- $\forall i \in I : \delta_n(s, i) \leftarrow \delta_n(s, i) \cup \delta_n(s', i)$
- $\forall s_t \in S, \forall i \in I :$ **replace** $s' \in \delta_n(s_t, i)$ **with** s
- **if** $s_c = s'$ **then** $s_c \leftarrow s$
- Finally, s' may be deleted as described in Section 3.10

Initially, we use this merging procedure in Line 5 where all states with the same output are merged together, i.e., $\exists s \in S : \forall s' \in S : s \neq s' \implies$ **merge** s' **into** s, which leaves us with a single state for each output $|S| = |O|$. We do this to prevent choosing a wrong target state with Algorithm 9 in Line 9, where unexpected behavior could occur should both states not allow for the same transitions. This may happen if we distinguished two states in an earlier iteration of Algorithm 11 but with an incomplete W_o-set such that we missed some state with the same output. Merging all states before the loop prevents this.

The second point where merging is used is in Line 18. Once we create a new state with Algorithm 12 we cannot be sure which transitions this specific state will have. After these transitions were explored in Line 17 we might find out that the same state, identical in transition outputs and distinguishers, already exists. In this case, the newly split state will be merged into the already existing one. If we would not re-merge the split states in Line 18 we would endlessly split off identical states, never realizing that the split state already exists and thus producing a chain of identical states.

In the procedure in Line 18, two states $s, s' \in S$ may only be merged if they have the same output, the same number of transitions to states with the same outputs and all responses to their distinguishers are identical. More formally, the newly created state in Line 16, s', may be merged into s iff

$$\begin{aligned} &\lambda(s) = \lambda(s') \\ &\wedge \forall i \in I : (|\delta_n(s,i)| = |\delta_n(s',i)| \wedge \forall s_t \in \delta_n(s,i), \exists s'_t \in \delta_n(s',i) : \lambda(s_t) = \lambda(s'_t)) \\ &\wedge D(s) = D(s') \end{aligned} \tag{3.10}$$

We only check for the outputs of the target states on the transitions because the existing state may already have transitions to more distinguished states compared to the newly created one. In this case, we still allow the merge.

Deleting States. A state s' may be deleted if it was merged as explained in Section 3.10 or in Line 21, where all states are removed that are no longer reachable via Algorithm 5 but are also not abandoned, i.e., states that have the following property:

$$\text{BFS}(s_c, \{s'\}, I, \delta_n, A) = (\emptyset, \textbf{None}) \wedge s' \notin A \tag{3.11}$$

This happens once all incoming transitions of any not fully distinguished state were all split off in Algorithm 12.

In any case, deleting the state s' simply requires removing all of its entries from the affected data structures of *context*:

- $S \leftarrow S \setminus \{s'\}$
- $\delta_n \leftarrow \{(s,i) \mapsto (\delta_n(s,i) \setminus \{s'\}) \mid s \in (S \setminus \{s'\}) \wedge i \in I\}$
- $\lambda \leftarrow \{s \mapsto \lambda(s) \mid s \in (S \setminus \{s'\})\}$
- $A \leftarrow A \setminus \{s'\}$

O is not affected as it is not possible to remove all states with a given output. Removing a state is only possible after a new state with that output was split off, therefore, at least one state with each output will remain even if it might be an abandoned one.

Similarly, s_c is also not affected as we navigate to the already fully distinguished state s in Line 19 before we delete any states in Line 21.

Example. To better visualize what Algorithm 11 precisely does, we present the smallest possible example in Figures 3.1, 3.2, 3.3 and 3.4. The drawing convention of Moore machines is explained in Section 2.4.

Figure 3.4 shows how the underlying SUL actually behaves, i.e., two states with the output b and three states in total, while Figure 3.1 shows the current model, which is seemingly non-deterministic at the moment as there are two transitions from State 1 on input x.

At the beginning of the algorithm *context* looks like this: $I = \{x\}$, $O = \{a, b\}$, $S = \{0, 1\}$, $\delta_n = \{(0, x) \mapsto \{1\}, (1, x) \mapsto \{0, 1\}\}$, $\lambda = \{0 \mapsto a, 1 \mapsto b\}$, $s_c = 0$, $R = \{a \mapsto \emptyset, b \mapsto \{x\}\}$ and $A = \emptyset$.

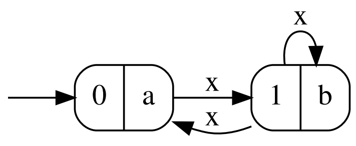

Figure 3.1: Initial hypothesis before distinguishing.

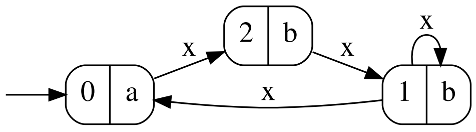

Figure 3.2: First iteration with newly split State 2.

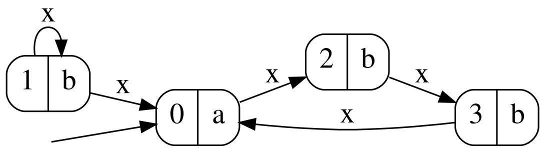

Figure 3.3: Second iteration with newly split State 3.

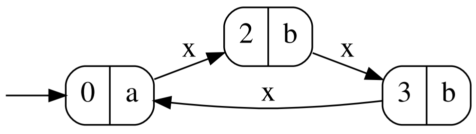

Figure 3.4: Resulting deterministic hypothesis.

Important to note is that we already calculated a distinguisher, namely x, for the output b as can be seen in R, which we can now use to distinguish State 1.

Starting the algorithm with Figure 3.1, we first initialize $D = \emptyset$. We can ignore Line 5, as there are no states with duplicated outputs in Figure 3.1, and start directly with the loop in Line 6. We enter the loop because there is one distinguisher for State 1 that has not yet been applied and recorded in D, i.e., $|R(\lambda(1))| > |D(1)|$, therefore we choose $s = 0$, $i = x$, and $s' = 1$ in Line 8. We do not have to navigate to State 0, because we are already there, and enter State 1 via the step in Line 11. Applying the distinguisher $d = x$ in Line 14 gives us the response sequence $\langle b \rangle$. We then split off a new State 2 in Algorithm 12 and add the observed response to D, which looks like $D = \{2 \mapsto \{x \mapsto b\}\}$, which results in Figure 3.2.

As Figure 3.2 is already input-complete, no exploration is required. Likewise, there are no identical states, i.e., 1 and 2 do not have the same entries in D, and there are no unreachable states, which lets us skip Line 18 and 21 respectively. We start the loop from the top again in Line 6, choosing $s = 2$, $i = x$, and $s' = 1$ in Line 8. Note that State 2 is fully distinguished as D has a corresponding mapping for every entry in $R(b)$ for State 2. We navigate to 2, apply step x and enter State 1, this time splitting off the new State 3 which results in Figure 3.3 with $D = \{2 \mapsto \{x \mapsto b\}, 3 \mapsto \{x \mapsto a\}\}$.

Subsequently, State 1 is now unreachable and will be removed in Line 21, which results in Figure 3.4. All states are now fully distinguished and we terminate Algorithm 11, returning the new *context*, which will in turn lead to the termination of the main algorithm once no counterexample is found by the equivalence oracle.

Termination. Algorithm 11 will terminate after a finite number of iterations in Line 6, if not earlier due to unexpected behavior:

Any not fully distinguished state s' with $m = |D(s')|$, $w = |R(\lambda(s'))|$ and $m < w$ will be chosen as the target s' in Line 8 exactly t_δ times, whereby t_δ is the number of incoming transitions to s'. This process may produce up to t_δ newly split states. However, if some of these split states have the same transition output behavior and the same responses to distinguishers then they are merged again in Line 18. Thus we only keep the number of states with different responses, which will never exceed the number of real underlying SUL states with that output as otherwise some of the responses to distinguishers must be identical, i.e., with n underlying states there cannot be more than n different responses on the same input sequence. Each of the remaining states would then have to be itself chosen as s' but with $m + 1$ compared to the original m of s'. At some point all remaining states will have their $m = w$ which leads to the termination of the algorithm.

4 Illustrative Examples

To better show the workings of the algorithm we provide some examples which show in detail how the algorithm can learn particular Moore machines.

We classify the examples according to some of the properties the underlying Moore machine M has that we want to learn:

- We say M is *trivial* if all of its states have unique outputs, i.e., $|O| = |S|$.
- Contrary M being *non-trivial* means that at least two states have the same output, i.e., $|O| < |S|$.
- A non-trivial example in which no states have a unique output will be called *indistinct* ie. $\forall s \in S, \exists s' \in S : s \neq s' \wedge \lambda(s) = \lambda(s')$.
- Finally, a *normal* example would be a non-trivial one which is not indistinct, i.e., has at least one state that has a unique output: $\exists s \in S, \forall s' \in S : s \neq s' \implies \lambda(s) \neq \lambda(s')$.

All provided examples are *input-complete* but may or may not be *strongly-connected.*

4.1 Example 1

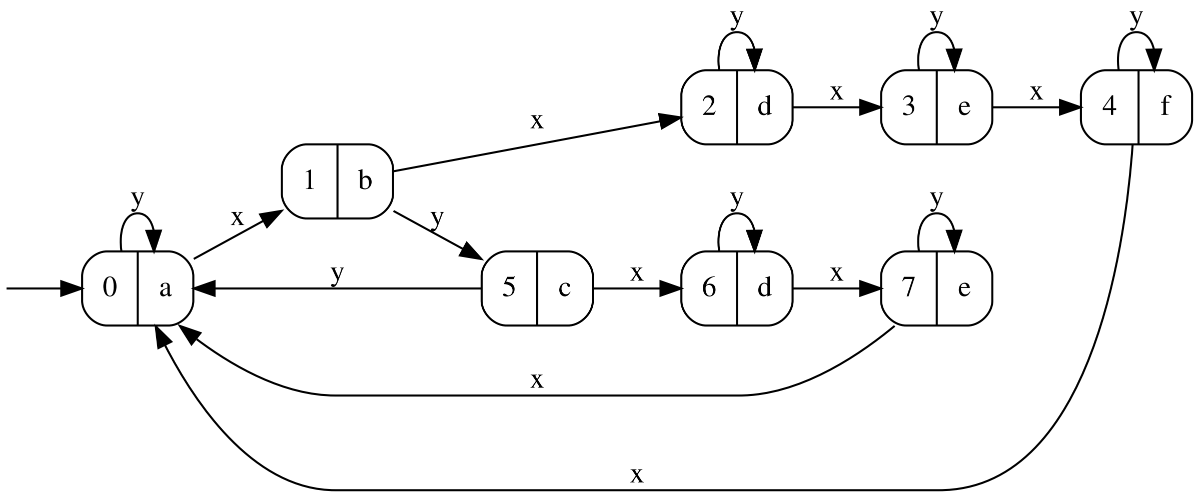

Figure 4.1: Moore Machine for Example 1.

Example 1 as shown in Figure 4.1 is a typical example that can be solved by the algorithm. It is *normal* as the two states 2 and 6 as well as 3 and 7 have identical outputs, d and e respectively, but there are also states with unique outputs. Furthermore, the example is *strongly-connected.*

We start the main algorithm (Algorithm 1) with the arguments $I = \{x, y\}$ and $\text{len}_{max} = 0$ as well as a SUL corresponding to Figure 4.1 and an appropriate equivalence oracle. The output of the initial state o_0 is a and the *context* is initialized as described in Line 4. In this example no states will be abandoned so A can be ignored as it will stay empty. Additionally, because $\text{len}_{max} = 0$, only the direct transfer sequences from Algorithm 4 will be used which is sufficient to learn this example.

Learning Round 1

Explore Missing. Table 4.1 shows the changes to *context* which occur during the first call to Algorithm 2 through Algorithm 8 which is itself called in Line 12 and in Algorithm 4.

We use $\Delta_+ X$ to denote an addition to the set, mapping or sequence X in a given step:

Table 4.1: Changes during the execution of the initial call of Algorithm 2 for Example 1.

Index	Input Step	Observed	Actual	s_c	$\Delta_+ O$	$\Delta_+ S$	$\Delta_+ \delta_n$	$\Delta_+ \lambda$	$\Delta_+ Tr$
0		a	0	s_0	a	s_0		$s_0 \mapsto a$	
1	x	b	1	s_1	b	s_1	$(s_0, x) \mapsto s_1$	$s_1 \mapsto b$	(a, x, b)
2	x	d	2	s_2	d	s_2	$(s_1, x) \mapsto s_2$	$s_2 \mapsto d$	(b, x, d)
3	x	e	3	s_3	e	s_3	$(s_2, x) \mapsto s_3$	$s_3 \mapsto e$	(d, x, e)
4	x	f	4	s_4	f	s_4	$(s_3, x) \mapsto s_4$	$s_4 \mapsto f$	(e, x, f)
5	x	a	0	s_0			$(s_4, x) \mapsto s_0$		(f, x, a)
6	y	a	0	s_0			$(s_0, y) \mapsto s_0$		(a, y, a)
7	x	b	1	s_1					(a, x, b)
8	y	c	5	s_5	c	s_5	$(s_1, y) \mapsto s_5$	$s_5 \mapsto c$	(b, y, c)
9	x	d	6	s_2			$(s_5, x) \mapsto s_2$		(c, x, d)
10	y	d	6	s_2			$(s_2, y) \mapsto s_2$		(d, y, d)
11	x	e	7	s_3					(d, x, e)
12	y	e	7	s_3			$(s_3, y) \mapsto s_3$		(e, y, e)
13	x	a	0	s_0			$(s_3, x) \mapsto s_0$		(e, x, a)
14	x	b	1	s_1					(a, x, b)
15	y	c	5	s_5					(b, y, c)
16	y	a	0	s_0			$(s_5, y) \mapsto s_0$		(c, y, a)
17	x	b	1	s_1					(a, x, b)
18	x	d	2	s_2					(b, x, d)
19	x	e	3	s_3					(d, x, e)
20	x	f	4	s_4					(e, x, f)
21	y	f	4	s_4			$(s_4, y) \mapsto s_4$		(f, y, f)

- $\Delta_+ Set \equiv Set \leftarrow Set \cup \{\Delta_+\}$
- $\Delta_+ Mapping \equiv Mapping \leftarrow Mapping \cup \{\Delta_+\}$
- $\Delta_+ Seq \equiv Seq \leftarrow Seq + \{\Delta_+\}$

In Line 11 of Algorithm 2 we decide to always choose $i \in I$ in alphabetical order, i.e., first the input x, if it has no transition, and only then y.

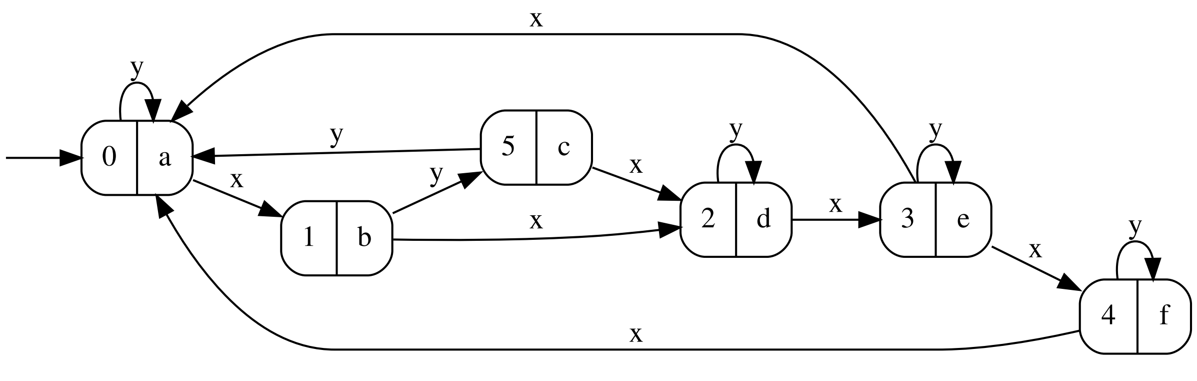

Figure 4.2: Non-deterministic hypothesis after initial `exploreMissing` for Example 1.

During the calls to `navigateToTarget` (Algorithm 4) on Line 7 the target state is either already reached or the direct word in Line 9 of Algorithm 4 succeeds in navigating to the target state. In fact, in this example navigation always succeeds with the direct word in Line 9 and navigating via words from the trace is not used as len$_{max}$ $= 0$.

After all steps in Table 4.1, all transitions have one transition for every input except for $\delta_n(s_3, x) = \{s_0, s_4\}$ which has two transition on the same input which can be seen in Figure 4.2, i.e., at this point δ_n is *non-deterministic* (see Section 3.3.2). This is the case because we wrongly assumed to be in State 2 after Index 9 on Table 4.1 when we actually visited State 6. From there we entered State 3 at Index 11 which was actually State 7, which lead to the additional entry for x at Index 13, i.e., looking at the additions to the trace from Index 4 and 13 will show the conflicting entries. Comparing Figure 4.1 and Figure 4.2 one can see that the states with duplicated outputs are merged in the hypothesis because of this wrong assumption, that we believed to have visited State 3 when in reality it was State 7. At this point, there was however no way for us to have known that these two states are different.

Because δ_n is input-complete, i.e., has at least one transition for every input, Algorithm 2 will terminate.

Next Algorithm 3 is called. However, as all states are reachable from the current State s_4 nothing changes and no states are abandoned.

Calculate W_o-sets. After Algorithm 2 finishes, we next calculate the W_o-sets in Algorithm 10 using the following arguments:

The trace Tr is the concatenation of each entry of the column Δ_+Tr in Table 4.1. As Algorithm 8 was called 21 times during Algorithm 2 that means the trace has 21 entries.

The mapping of W_o-sets to their responses, R, was not explicitly tracked in Table 4.1 but it can be easily seen that R only changes in Line 14 in Algorithm 9 every time a new output is observed. Consequently, all W_o-sets in R are still empty, i.e., $R = \{o \mapsto \emptyset \mid o \in O\}$.

Calculating C from Line 3 to Line 8 will result in a set of all pairs of indices which are identical in source output and input but different in output of the destination state. For the current example this set will be: $C = \{(4, 13), (13, 20)\}$. Taking a look at the entries Δ_+Tr at indices 4 and 13 in Table 4.1 one can see that they are (e, x, f) and (e, x, a) respectively. These pairs of indices in C are the "visible" differences that could also be directly observed from the graph in Figure 4.2.

We only have a single state with output e, namely s_3, in our *context*. With the same input x two different target states (s_4 and s_0) were seemingly reached with output f and a respectively. This *seeming* non-determinism results from the fact that s_3 must be composed of at least two states with the same output instead of a single state. These states can be at least partially distinguished with the input x such that one state always transitions to s_4 and another to s_0. This distinguisher will be recorded in R like so: $R = \{a \mapsto \emptyset, b \mapsto \emptyset, c \mapsto \emptyset, d \mapsto \emptyset, e \mapsto \{\langle x \rangle\}, f \mapsto \emptyset\}$.

Distinguish. Algorithm 11 is now executed with the distinguisher calculated in Section 4.1. Because we only have a distinguisher for output e and not yet for d we cannot produce the correct hypothesis in this learning round. This example was chosen specifically because the shortest transfer sequence to a state with output d is always shorter via state input x from State 1 than from State 5. Therefore, the transition to State 6 in Figure 4.1 will not be taken at all during this execution, which will result in us not encountering any unexpected behavior in this round.

All 27 steps taken during this execution of Algorithm 11 are recorded in Table 4.2 with comments about where exactly in the algorithm the step was produced.

Initially, we navigate to State $s = s_2$, which was chosen in Line 8, where we enter $s' = s_3$ in Line 11. This corresponds to the Indices up to 25 in Table 4.2. Then we apply the distinguisher x, which leads us to State s_4 after which we split off a new State s_6 with $D(s_6) = \{x \mapsto f\}$. The newly split State s_6 is still missing the transition on y, so we navigate back to it, explore it and find the transition on y to s_3 in Index 31. The reason why we chose s_3 as the target and not s_6 is explained in detail in Section 3.8 in Case 3. This now results in the hypothesis in Figure 4.3.

Next we do the same again, this time choosing $s = s_6$ in Line 8, as s_6 is now fully distinguished, and $s' = s_3$, which we enter via input y in Index 37, and split off a new State s_7 with $D(s_7) = \{x \mapsto f\}$.

Table 4.2: Changes during the execution of Algorithm 11 for Example 1.

Index	Input Step	Observed	s_c	Comment	$\Delta_+ S$	$\Delta_+ \delta_n$	$\Delta_+ \lambda$	$\Delta_+ Tr$
22	x	a	s_0					(f, x, a)
23	x	b	s_1					(a, x, b)
24	x	d	s_2					(b, x, d)
25	x	e	s_3	apply x				(d, x, e)
26	x	f	s_4	split s_3	s_6	$(s_6, x) \mapsto s_4$	$s_6 \mapsto e$	(e, x, f)
27	x	a	s_0					(f, x, a)
28	x	b	s_1					(a, x, b)
29	x	d	s_2					(b, x, d)
30	x	e	s_6					(d, x, e)
31	y	e	s_3	explore s_6		$(s_6, y) \mapsto s_3$		(e, y, e)
32	x	f	s_4					(e, x, f)
33	x	a	s_0					(f, x, a)
34	x	b	s_1					(a, x, b)
35	x	d	s_2					(b, x, d)
36	x	e	s_6					(d, x, e)
37	y	e	s_3	apply x				(e, y, e)
38	x	f	s_4	split s_3	s_7	$(s_6, y) \mapsto s_7$	$s_7 \mapsto e$	(e, x, f)
39	x	a	s_0					(f, x, a)
40	x	b	s_1					(a, x, b)
41	x	d	s_2					(b, x, d)
42	x	e	s_6					(d, x, e)
43	y	e	s_7					(e, y, e)
44	y	e	s_3	explore s_7		$(s_7, y) \mapsto s_3$		(e, y, e)
45	x	f	s_0	merge s_7				(e, x, f)
46	x	a	s_0					(f, x, a)
47	x	b	s_0					(a, x, b)
48	x	d	s_0					(b, x, d)
49	x	e	s_0					(d, x, e)

Again we need to explore the missing transition on y, which we do in Index 44 in Table 4.2, but this time s_6 and s_7 are equivalent according to Definition 3.10, which allows us to immediately merge s_7 into s_6, deleting s_7 in the process. This now leaves the hypothesis looking like in Figure 4.4.

Finally, we remove the now unreachable State s_3 which leaves the hypothesis, which is subsequently returned to the main algorithm, just as in Figure 4.5.

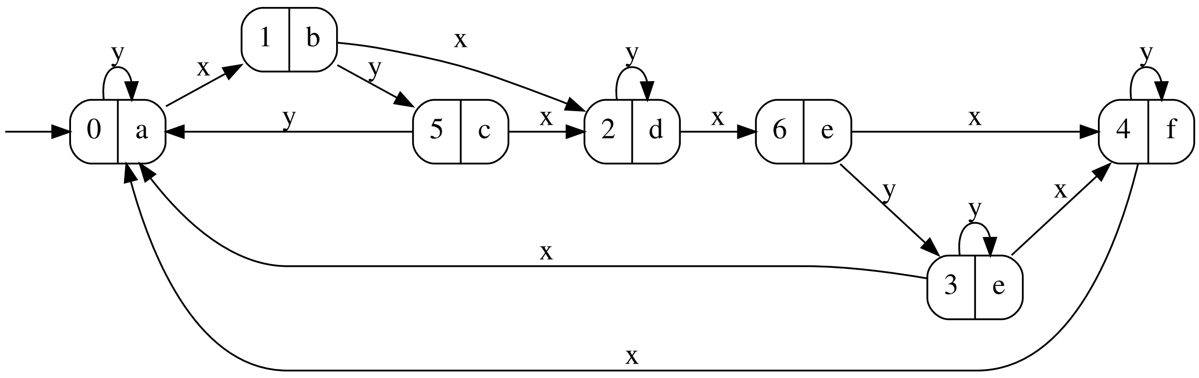

Figure 4.3: Hypothesis after first loop of Algorithm 11 for Example 1.

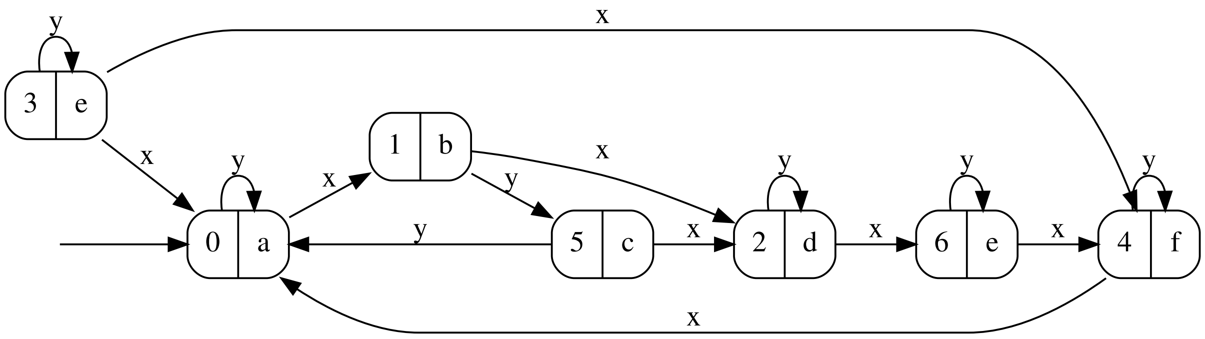

Figure 4.4: Hypothesis in second loop of Algorithm 11 in Line 20 before removing unreachable states for Example 1.

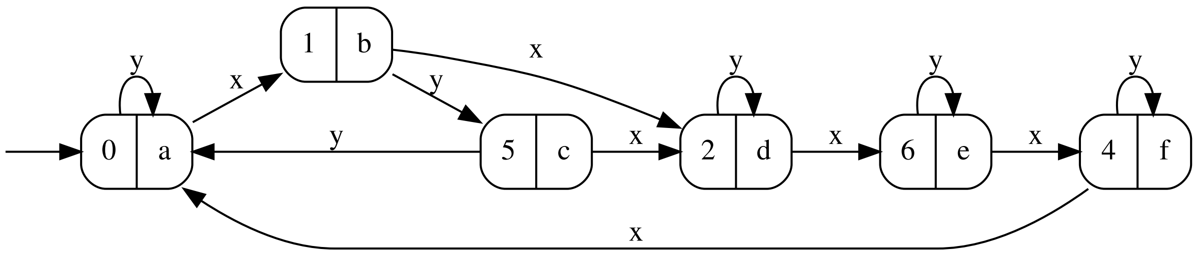

Figure 4.5: Hypothesis after completing Algorithm 11 for Example 1.

Learning Round 2

We enter the second learning round, iteration two of Algorithm 1, with the result from Algorithm 11 in Figure 4.5. No state exploration is performed because δ_n is input-complete and we perform an equivalence query, in Line 15, because δ_n is deterministic. We assume that the equivalence oracle produces the steps shown in Table 4.3. At Index 56 we encounter unexpected behavior due to finally finding the transition via State 5 again. This leads to the immediate termination of the equivalence query and the start of the next learning round.

Table 4.3: Changes during the equivalence query for Example 1.

Index	Input Step	Observed	Actual	s_c	Δ_+O	Δ_+S	$\Delta_+\delta_n$	$\Delta_+\lambda$	Δ_+Tr
50	x	f	4	s_4					(e, x, f)
51	x	a	0	s_0					(f, x, a)
52	x	b	1	s_1					(a, x, b)
53	y	c	5	s_5					(b, y, c)
54	x	d	6	s_2					(c, x, d)
55	x	e	7	s_6					(d, x, e)
56	x	a	0	s_0			$(s_6, x) \mapsto s_0$		(e, x, a)

Learning Round 3

Calculate W_o-sets. After the equivalence query in Round 2 our hypothesis now looks like Figure 4.2 again with the difference that State 3 is now replaced with State 6. However, during the distinguishing in Section 4.1 in Round 1 and and the equivalence query in Section 4.1, we acquired a longer trace, which will now produce new distinguishers in Algorithm 10:

The set C is produced first, which we split for demonstration purposes like this:

- $C_1 = \{(4, 56), (20, 56), (26, 56), (50, 56)\}$
- $C_2 = \{(13, 32), (13, 38), (13, 45)\}$
- $C_3 = \{(13, 50), (13, 26), (32, 56), (38, 56), (45, 56)\}$

where $C = C_1 \cup C_2 \cup C_3$.

Note that the elements of preceding C-sets, like $(4, 13)$ or $(13, 20)$ from Section 4.1, are not included in the current set C due to the optimization in Section 3.9.

C_1 contains the pairs of indices that produce the distinguisher $\langle x, x\rangle$ for output d, and consequently $\langle x\rangle$ for output e, C_2 contains the pairs that produce the distinguisher $\langle x, y, x\rangle$ for output d, and consequently $\langle y, x\rangle$, and C_3 contains the pairs that produce only $\langle x\rangle$ for output e.

This leaves us with a mapping R like this:

$R = \{a \mapsto \emptyset, b \mapsto \emptyset, c \mapsto \emptyset, d \mapsto \{\langle x, x\rangle, \langle x, y, x\rangle\}, e \mapsto \{\langle x\rangle, \langle y, x\rangle\}, f \mapsto \emptyset\}$.

Distinguish. As we now have distinguishers for outputs d in addition to e the states with output d will be distinguished first, as we require a fully distinguished state as s in Line 8, before the respective succeeding state with output e can be distinguished. Algorithm 11 will produce the hypophysis in Figure 4.6 after 16 iterations and 196 steps. Even one distinguisher per output would have been sufficient in this case, however, because we have two per state we require more iterations.

Learning Round 4

The hypothesis at this point in time looks like Figure 4.6. As δ_n is deterministic an equivalence query will be performed. Because Figure 4.6 is equivalent to the original Moore machine in Figure 4.1 no counterexample or unexpected behavior is found, thus the algorithm terminates with the correct result.

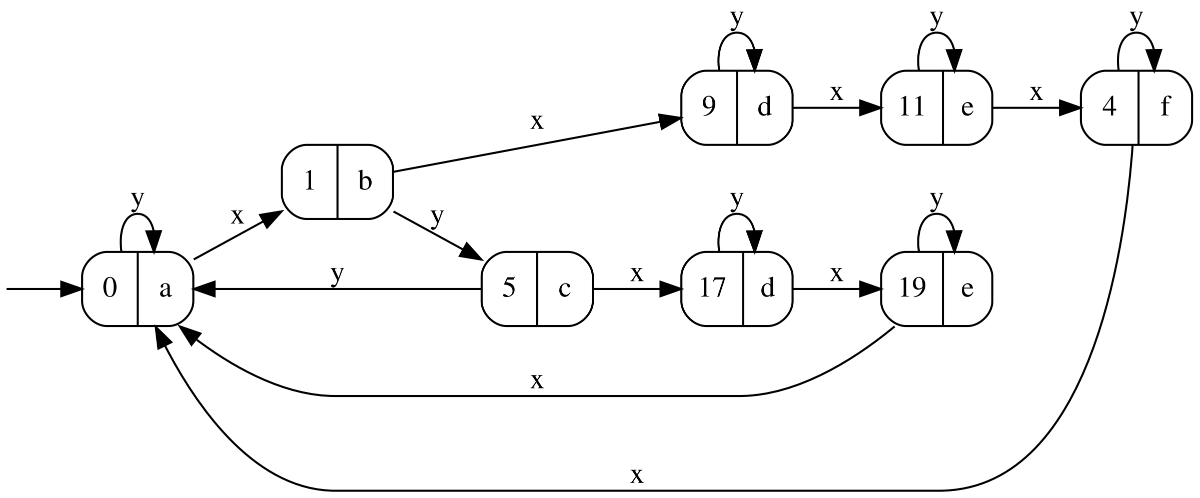

Figure 4.6: Completely learned hypothesis for Example 1.

4.2 Example 2

Example 2 is *normal* and also the smallest model that requires navigation via Algorithm 6 in order to correctly learn the model. In fact, this example will successfully terminate with $len_{max} \geq 1$ but will run into the alternative endless loop prevention in Line 7 in Algorithm 1 if $len_{max} = 0$. The example Moore machine is shown in Figure 4.7.

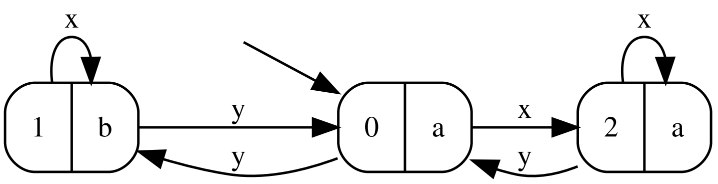

Figure 4.7: Moore Machine for Example 2.

Table 4.4 shows the first 11 steps taken for Example 2. Initially, we do not even discover State 1 without the equivalence oracle, which we query at Index 3. In Iteration 2 of Algorithm 1 we then correctly find the distinguisher for output a, namely $\langle y \rangle$, and apply it on State s_0 at Index 7. However, here is were the problem occurs. Due to how the splitting works we end up confusing states 2 and 0 which, when navigating back to State s_1, results in us not reaching s_1 with the direct transfer sequence of $\langle y \rangle$ at Index 11.

There are two possible scenarios now depending on len_{max}. If $len_{max} = 0$ then not reaching State s_1 leads to abandoning s_1 in Line 11 of Algorithm 4 which in turn leads us to terminating Algorithm 11 in Line 19. We would then run into the exact same sequence of step again and again, however, after one more iteration, in which the distinguishing fails in precisely the same way, the algorithm terminates with the alternative condition in Line 7 in Algorithm 1. In reaction to such a termination len_{max} can be increased.

Table 4.4: Changes during first two iterations of Algorithm 1 for Example 2.

Index	Input Step	Observed	Actual	s_c	Comment	$\Delta_+ S$	$\Delta_+ \delta_n$	$\Delta_+ Tr$
0		a	0	s_0	init	s_0		
1	x	a	2	s_0	explore		$(s_0, x) \mapsto s_0$	(a, x, a)
2	y	a	0	s_0			$(s_0, y) \mapsto s_0$	(a, y, a)
3	y	b	1	s_1	eq-query	s_1	$(s_0, y) \mapsto s_1$	(a, y, b)
4	x	b	1	s_1	explore		$(s_1, x) \mapsto s_1$	(b, x, b)
5	y	a	0	s_0			$(s_1, y) \mapsto s_0$	(b, y, a)
6	y	b	1	s_1	distinguish			(a, y, b)
7	y	a	0	s_0	apply y			(b, y, a)
8	y	a	1	s_1	split s_0	s_2	$(s_1, y) \mapsto s_2$	(a, y, b)
9	y	a	0	s_2				(b, y, a)
10	x	a	2	s_0	explore s_2		$(s_2, x) \mapsto s_0$	(a, x, a)
11	y	a	0	s_0	navigate s_1			(a, y, a)

The second scenario is with $len_{max} \geq 1$. After failing to reach State s_1 directly in Line 19 additional transfer sequences are generated from the trace with Algorithm 6. In this case, $\langle y \rangle$ is found as a transfer sequence from a state with output a to b, e.g. at Indices 3, 6 and 8 in Table 4.4, which is then tried after Index 11 as the next step. Because $\langle y \rangle$ is now applied from State s_0, we reach s_1 and may continue with the distinguishing process, which eventually results in a correct hypothesis after three learning rounds and 121 steps.

This example is very small but still illustrates how sometimes, through confusion of states, navigation does not always succeed using the shortest transfer sequence. In such cases additional potential transfer sequences from the trace may help to find target states again.

5 Extension: Time-Triggered Behavior

5.1 Time-Triggered Behavior

Some of the measurement devices that the algorithm was developed for could not be modeled only as Moore machines. It turns out that some of their states have transitions that are taken after a certain amount of time passed *without* any input being required to trigger the transitions. We call such behavior *time-triggered*, as the transition is not triggered by an input to the SUL but instead by the elapsing of a certain amount of time. Furthermore, we call a state *timed* if it exhibits such time-triggered behavior.

Usually, timed states on the measurement devices are actions, such as measurements or checks, that require a certain amount of time. The state is entered when the measurement is started and the state is left again after a certain period of time. During the measurements most other inputs are mostly ignored as we will see during the case study of the Smoke Meter in Section 6.2. Enabling our algorithm to deal with such behavior was important for learning the more complex measurement devices, which led to this extension of the base algorithm.

Such time-triggered behavior for automotive measurement devices was already encountered by Aichernig et al. [3] under similar circumstances, where it was dealt with in a more elaborate fashion using sink states. In our case, as will be explained below, we will primarily ignore other inputs in timed states and thus learn a partial model instead of a complete one.

Issues. Experiencing time-triggered behavior during the learning process can lead to a number of problems that might range in their severity from negligible up to the termination of the algorithm via one of the alternative termination conditions described in Section 3.4.

For one, time-triggered behavior introduces non-determinism into the learning process. Some transitions are taken only due to timing race conditions, depending on delays in messages or in the learning process. The main problem is that we receive the result of an output query at a time where we believe to be in a certain state, which might have changed earlier. Two examples of possible race conditions are demonstrated in the sequence diagrams in Figures 5.2 and 5.3.

Additionally, recording wrong transitions in the trace, due to time-triggered behavior, may lead to the generation of redundant distinguishers in Algorithm 10. Not only will this unnecessarily increase the runtime of the algorithm, but worse, some distinguishers may be generated for states that would otherwise have unique outputs. This may lead to the termination of Algorithm 11 in Line 7, if too many outputs wrongly gain distinguishers in this way.

Attempting to learn machines with time-triggered behavior with only the base algorithm usually resulted in either the premature termination of the algorithm due to lack of outputs without distinguishers or the generation of ever more distinguishers which slowed the algorithm significantly. Furthermore, the nature of race conditions would make it hard or even impossible to recreate earlier runs as the timings may be different between each learning process, such that we could not guarantee that the outcomes of different runs would be the same.

Solution. The solution we chose to deal with time-triggered behavior is twofold:

Firstly, we introduce a special input, the *waiting input* explained below in this Section, which is not passed to the SUL but instead results in a waiting action by the output oracle, i.e., when the output oracle receives this input it should actually wait for a certain amount of time. This input is added the input alphabet and signifies if states have time-triggered transitions or not.

Additionally, to actually detect state changes, during the waiting process the output oracle should periodically poll the current state of the SUL in order to detect when a state change actually happens time-triggered.

Secondly, we decided to not fully explore timed states with provided inputs because the trying of said inputs in a timed state may produce the issues described above. Instead, timed states will, when entered, only be left via their time-triggered transition and no other inputs are performed in them.

Not fully exploring some of the states will result in a *partial model*, such as the one in Figure 5.1. A partial model is a FSM that is not input-complete and may be missing transitions for multiple inputs on some states. In our case, all timed-states have only a single outgoing transition on their *waiting input*.

This prevents us from purposefully exploring inputs in states that might be interrupted by their time-triggered transitions to other states and, in addition to the detecting of state changes during waiting, enables us to mostly deal with time-triggered behavior. The limitations of this approach are described below.

The downside of this approach is that we only produce a partial model anymore, meaning that we do not explore the entire system but only a part of it. It also limits us in choice for testing algorithms, as described in Section 5.2, and requires some changes to the base algorithm, as detailed in Section 5.2.

Waiting Input. A specific input i_t, which we subsequently call *waiting input*, which may not be part of the normal input alphabet, is chosen before the algorithm is started. This input i_t is then added to the input alphabet. If this special input is received by the output oracle in Figure 2.1 it is not forwarded to the SUL.

Instead, upon receiving i_t, the output oracle starts waiting for a specified amount of time, t_{max}, while also polling the output of the current state from the SUL for changes. If during the waiting process a change in output is detected by the output oracle then the waiting is stopped and the changed output is returned from the output query. Otherwise, if the output does not change, the current output is returned after t_{max} time. An example depicting this behavior is also shown in Figure 5.4.

In short, the waiting input corresponds to the action of waiting for a certain amount of time, or a change of outputs, instead of a real input provided to the SUL.

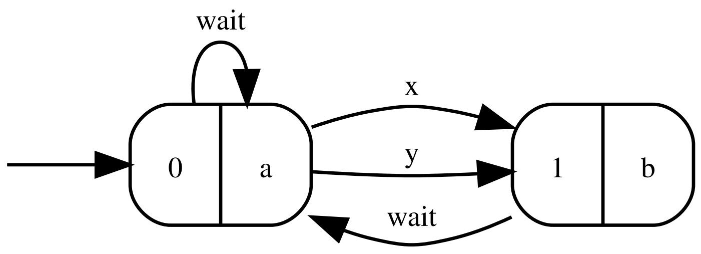

Figure 5.1: Partial Moore machine with waiting input "wait" where State 1 is timed.

In Figure 5.1 we see a small example of a partial Moore machine with waiting input i_t = wait. As can be seen, the waiting input of State 1 leads to a different state, namely State 0, which makes it a timed state. Furthermore, because State 1 is timed, no other inputs are accepted in it, which makes this Moore machine *partial* as it is not input-complete. State 0 also has the waiting input on a transition, however, it is not a timed state as the waiting input leads to a self transition. This corresponds to no change in output if we wait in this state, i.e., the definition of a normal state.

Example without Waiting Input. The sequence diagram in Figure 5.2 shows a possible race condition during the exploration process of Figure 5.1 if the waiting input is not added or at least not performed first. For this example, we imagine that inputs x and y are ignored in State 1, which we would interpret as self transitions during the learning process, and that the transition back to State 0 is triggered after a fixed time interval t_w. The sequence diagram also shows that inputs and outputs as well as state changes are not instantaneous. In the theoretical model time is ignored, in practice, however, tiny delays in state changes and in execution speed of the algorithm itself can change the behavior of the system as shown here.

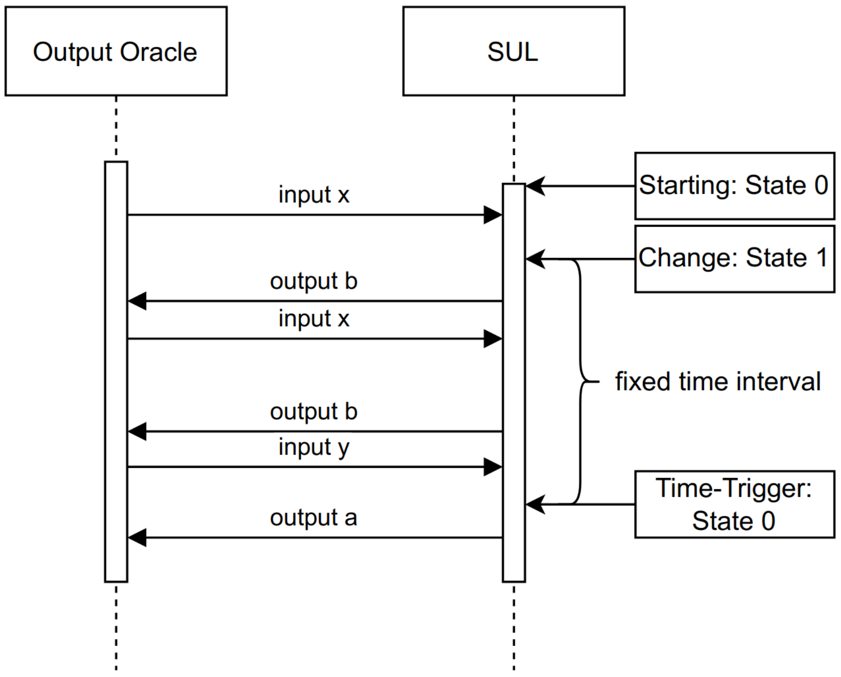

Figure 5.2: Sequence diagram of a possible interaction between output oracle and SUL during exploration *without* waiting input of Figure 5.1, triggering before output.

Starting in State 0 we reach State 1 with the input x, to which the SUL returns the output b of State 1. The moment we enter State 1 a process is started that takes a certain amount of time to finish, t_w. Once t_w time has passed we change back to State 0 via a time-triggered transition. In this example, we continue sending inputs to the SUL, which are ignored in State 1, which leads us to wrongly assume that x is a self transition and y is a transition back to State 0. This conclusion is entirely based on the order in which the inputs were tried. If we had attempted input y first and x second we might have concluded that x led back to State 0 instead.

There is, of course, no guarantee that subsequent visits to State 1 would result in the same transitions. Dependent on the time delay between messages and the order in which inputs are performed on the states, different observations would be made which would result in either a wrongly learned model, the premature termination of the algorithm or, in the worst case, an endless loop.

The sequence diagram in Figure 5.3 shows an even more problematic race condition in the timing of the time-triggered state transition. In this case, we still believe to be in State 1 when we send the input y. However, the time-triggered state change has already happened, which means that input y could have led us to a completely different state, which would have resulted in potentially multiple incorrect transitions.

Example with Waiting Input. In Figures 5.2 and 5.3 we did not have to make a distinction between the learner and the output oracle, as there was no difference between their received inputs and outputs, i.e., all inputs were passed directly through to the SUL and all outputs were passed directly back to the learner. In Figure 5.4 we now show the successful identification of State 1 as timed, which now requires the differentiation between learner and output oracle.

The learner uses the waiting input $i_t =$ wait exactly like any other input, with the exception that it is tried first as explained in Section 5.2. On the other hand, the output oracle does not pass the waiting input to the SUL. Instead, it waits for a maximum waiting time t_{max} and polls the current state of the SUL instead. The polling is the periodic sending of the empty input ϵ, to which Moore machines will respond with the output of the current state without changing it. While the output stays the same, in this

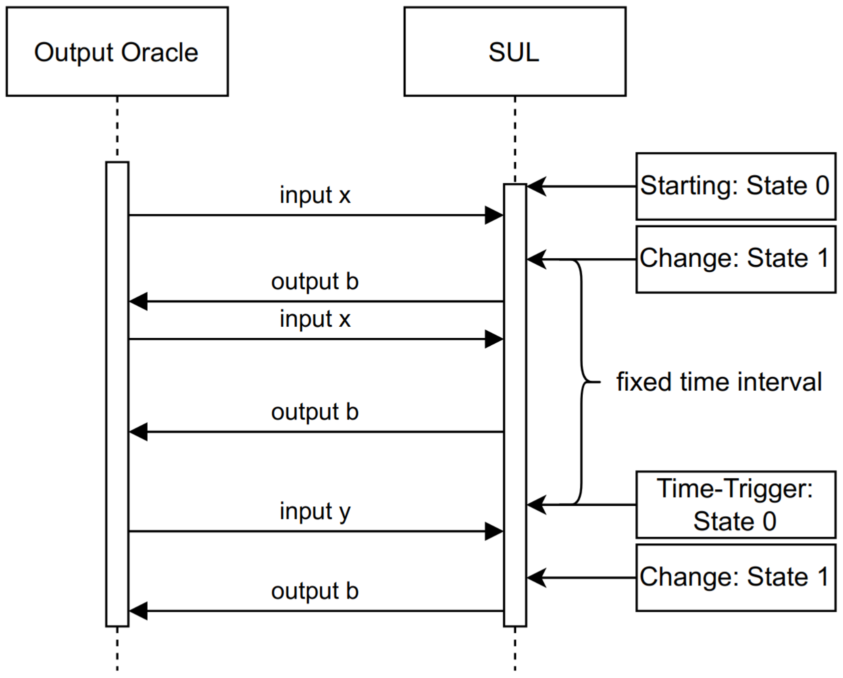

Figure 5.3: Sequence diagram of a possible interaction between output oracle and SUL during exploration *without* waiting input of Figure 5.1, triggering before input.

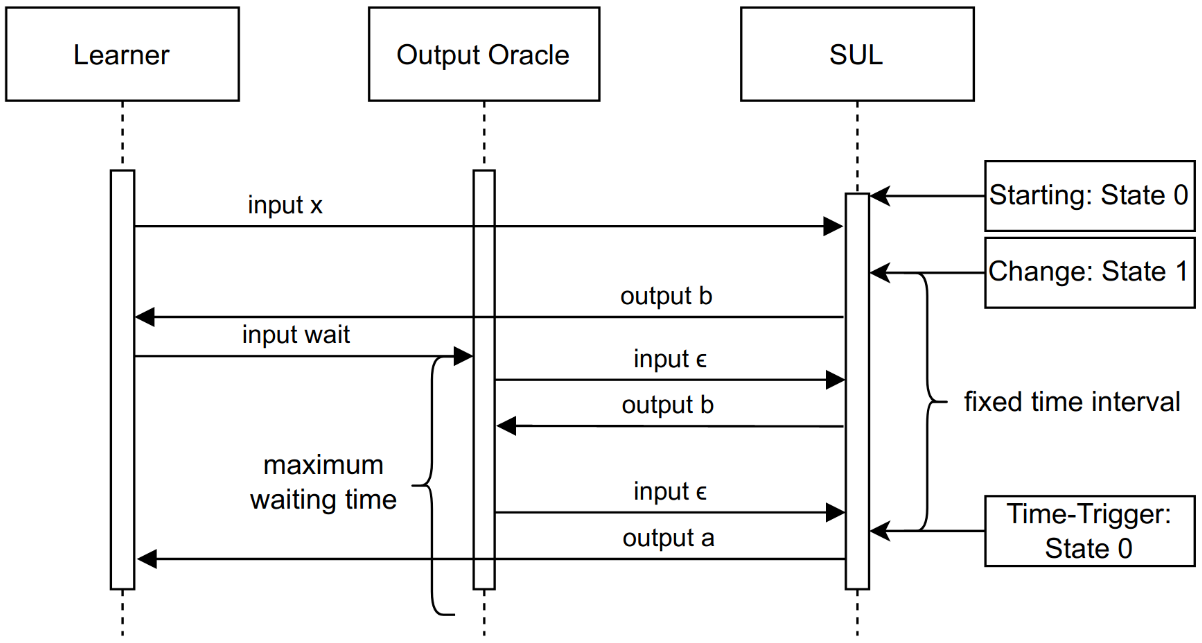

Figure 5.4: Sequence diagram of a possible interaction between learner, output oracle and SUL during exploration *with* waiting input of Figure 5.1, detecting State 1 as timed.

case, output b was the last observed output to the initial input x, we continue polling the state. Only if we receive a new output, in this case a, do we stop the waiting prematurely and return the new output to the learner.

Alternatively, if we do not observe any changes in input during the waiting period, we return after the maximum waiting time t_{max} expires with the last observed output.

Limitations. It is clear from the example in Section 5.1 how the incorrect transitions can occur, if the time-triggered behavior is not dealt with. However, the extended method's success depends on a number of variables that we have to choose carefully before the algorithm starts:

The maximum waiting time, t_{max}, which can be seen in Figure 5.4, represents the longest amount of time the output oracle will wait for a state change. It is quite intuitive that we can only catch time-triggered behavior that we actually wait for, meaning that if we choose t_{max} to be smaller than the longest actual time-triggered waiting time t_w we cannot detect it. This may in turn lead to all the issues, described in Section 5.1, which we wanted to prevent with this approach in the first place. In contrast, we also do not want to choose t_{max} to be too large, as this would increase the runtime of the algorithm because we have to wait for up to t_{max} on every waiting input transition. An additional complication may also be that the actual waiting time for the same transition varies. In that case, we have to choose t_{max} to still catch the longest outliers in waiting times.

The polling interval, t_p, is another variable that must be chosen with care. This is the time between receiving an output from the SUL and querying the current state during waiting. If t_p is too large, i.e., there are long waiting times between state queries, then it might happen that we miss an entire time-triggered transition. The exact scenario may start with the query of the output during waiting, which is not changed, then in the time it takes the output oracle to query the state again, we have two time-triggered state changes for which both take less time than t_p. To prevent such a scenario, we recommend to choose t_p as small as feasible. However, if the time-triggered transitions of the system are very fast then this may still happen even with a small t_p, as t_p may be bound by network speed or other hardware limitations. There is a bound on the minimum required sampling rate depending on the bandwidth of the connection that was established by Shannon [35] that can be determined for such types of communication.

A fundamental limitation of this approach is that, due to only learning partial models, we do not explore the entire behavior of the system anymore. One possible solution for working around this is to explore the unused inputs outside of the learning process. For example, we used fuzzing methods on the partial hypothesis *after* we finished the learning process to discover more about the behavior of other inputs in timed states.

Additionally, learning only partial models restricts us in our choice of testing algorithm for the equivalence oracle or at least requires us to modify the testing algorithms beforehand, as described in Section 5.2.

In the future, some of the problems encountered due to time-triggered behavior could probably be resolved by modeling the SUL not as a Moore machine but as a *timed automata*, as proposed by Alur and Dill [9] and in the Uppaal model checker by Behrmann et al. [12]. We describe this possibility further in Section 7.4. Of course, this would probably require the redesign of the entire algorithm, as it was constructed specifically with Moore machines in mind.

5.2 Modifications to the Algorithm

In order to handle time-triggered behavior as described in Section 5.1 some changes had to be made to the oracles, introduced in Section 2.2, as well as to the base algorithm, presented in detail in Chapter 3.

Definition Timed State. We formally define the concept of a *timed state*, as described in Section 5.1, for a given waiting input i_t as

$$s \in S \text{ is timed} \iff \exists s' \in \delta_n(s, i_t) : s' \neq s \tag{5.1}$$

In short, states that have a transition to a different state on waiting input i_t are timed, while states with a self transition on said input are considered normal states, e.g., in Figure 5.1 State 1 is timed in contrast to State 0.

Redefining Input-Completeness. We earlier defined input-completeness of a state, in Definition 3.4 in Section 3.3.3, as having at least one transition to a non-abandoned state for every input $i \in I$ in the transition function δ_n. We chose this definition because we required our hypothesis to accept any input in every state and thus had to explore every single transition in Algorithm 2.

However, now that we are learning a partial model, which is by definition not input-complete, and do not want to explore other inputs in timed states, we provide the following definition for input-completeness instead

$$s \in S \text{ is input-complete} \iff s \text{ is timed} \lor \forall i \in I, \exists s' \in \delta_n(s,i) : s' \notin A \tag{5.2}$$

The definition for input-completeness stays the same for states that are not timed, however, all timed states are automatically considered input-complete, as we do not want to explore any other inputs in said states. In short, for the purpose of the algorithm, timed states are treated as input-complete.

This definition is used in Algorithm 2 in Line 4 where targets are chosen for exploration of inputs. With the new stipulations regarding input-completeness in Definition 5.2 we now do not target timed states for exploration anymore, which may leave these states only partially explored as described in Section 5.1.

Trying Waiting Input First. In order to determine if a state is timed, and to make sure to not run into the issues described in Section 5.1, we must perform the waiting input i_t first before any other input is tried. This is especially important for Line 11 in Algorithm 2 where all inputs are tried one after the other. We must now always choose i_t first, if the rest of the condition in Line 11 holds, and only then choose the other inputs. If we did not try the waiting input first then we might run into the same issues that we wanted to prevent, as we can only know if a state is timed once we performed the waiting input, similar to the examples shown in Figures 5.2 and 5.3.

No Other Inputs in Timed States. Once a state is identified as timed we do not want to perform any other inputs in that state except the waiting input. We already described the changes necessary to prevent the purposeful exploration of other inputs in Algorithm 2 above.

Furthermore, we are already protected against accidentally performing other inputs in timed states due to the concept of unexpected behavior. If during navigation in Algorithm 7 we enter any state unexpectedly then the navigation is immediately halted before any other input can be performed in Line 7. This prevents us from stubbornly performing input queries even though we entered a timed state.

Additionally, because timed states only have a single transition, all words purposefully navigating through timed states will only use the waiting input, as that is the only explored input available for Algorithm 5 and 6 to find.

Timing Aware Equivalence Oracle. Similar to how no other input may be tried in a timed state during the algorithm, they may also not be performed during an equivalence query. In fact, trying a different input in a timed state would lead to unexpected behavior in Algorithm 9, which would in turn terminate the equivalence query. Additionally, it would also lead to all the issues described in Section 5.1. This means that the testing algorithms have to apply the rule from Section 5.2, namely, that no other inputs may be performed in timed states. In other words, the testing algorithm must be able to work with a partial hypothesis.

A strong point of the base algorithm is that any conventional testing algorithm may be used by the equivalence oracle, as stated in Section 3.5. With these restrictions, that the hypothesis is partial and that only the waiting output may be performed in timed states, this is not the case anymore for the extended algorithm. Most testing algorithms would have to be adapted to follow these rules, while some might not be suitable for testing under these conditions.

In our examples with time-triggered behavior, such as the Smoke Meter in Section 6.2, we used a modified random walk equivalence oracle. A random walk oracle usually performs a certain number of steps with completely random inputs and has a certain chance to reset after every step. The modified version simply uses Definition 5.1 on every state in the hypothesis to detect if it is timed and only allows the waiting input on timed states. For all other states random inputs are still chosen from the input alphabet.

Timing Aware Output Oracle. The heart of this extension lies the correct interpretation of the waiting input by the output oracle as described in Section 5.1, namely, that the waiting input is not forwarded to the SUL but instead the state of the SUL is polled and that the query returns the last output either after a state change or after the maximum waiting time t_w elapsed. An example of this behavior is shown in Figure 5.4.

How this behavior is implemented may vary. Either the output oracle itself provides this functionality or a wrapper for the output oracle does. In any case, the following three parameters have to be known by the output oracle before the learning process is started, namely the waiting input i_w, the maximum waiting time t_w and the polling interval t_p. The choice of t_w and t_p is pivotal for the success of the extended algorithm as choosing either t_w or t_p to be too large or too small may result in a number of issues and limitations, as explained in Section 5.1. By contrast, the only important restriction about the choice of i_w is that it may not be contained in the normal input alphabet.

Leaving Timed States. In order to deal with the time-triggered behavior as described in Section 5.1 we must actively be waiting for a state change with the timing input. Otherwise, if we are "not looking" at a time were a state change happens, then we will miss it, which would lead to all the particular problems described above. To prevent us from accidentally missing a state change while we are doing an intense computation, and not currently interacting with the SUL, we may navigate to a known not-timed state before starting the computation.

In practice, Algorithm 10 turned out to be the only place where such an exception occurred. It is the most computationally expensive task, which also does not interact with the SUL. Especially as the trace gets longer its runtime might increase to the point where we would leave states via their timed-triggers while doing the computations. A detailed explanation of the runtime can be found in Section 3.9. We would like to run Algorithm 10 while the SUL is not in a timed state. This means that in Algorithm 1 in Line 11 before we start Algorithm 10 we determine all not-timed states, i.e., $\{s \mid s \in (S \setminus A) \land \neg(s \text{ is timed})\}$, and navigate to any one of them by using Algorithm 4 if possible.

Record Timings. Additionally to the mandatory modifications in Section 5.2, the learner may want to time each step performed and record the timings, either in the trace or in a separate data structure. This information can later be used to enhance the partial model with data points of interest, such as the average time taken or the standard deviation on each time-triggered transition. These can be analyzed to make statements about the length of the waiting period in general and about the regularity of the timings, i.e., do they vary wildly or are they stable, in particular.

Caching Timing Results. A last optional change for this extension is the caching of outputs for which the waiting time t_{max} elapsed without any state changes. If an output was observed that had no state change on the waiting output then future waits on that output can be returned immediately. This has the advantage of potentially increasing the speed of the learning process by a lot, because states that are known not have time-triggered behavior do not require any waiting anymore. However, it has the potential downside of failing to differentiate between timed and not-timed states with the same output. If there are multiple states with the same output, they must either all be timed or not, but not a mixture of both.

In other words, caching should not be done if there are multiple states with the same output of which some are timed and others are not. Furthermore, it should only be implemented if some knowledge about this property of the SUL is already available.

In case of the Smoke Meter in Section 6.2, it turned out that all states with the same outputs also have the same timing, such that this optimization was possible. We did, however, also perform the learning process without caching for all devices we tested.

6 Evaluation

6.1 Performance of Base Algorithm

Evaluating the performance of the algorithm is not as straight forward as with some other learning algorithms, primarily because it is so dependent on the exact shape of the model, but also because there might appear be sizable outliers even among models with the same number of states and unique outputs. We tested the algorithm with different combinations of number of states, unique outputs and len_{max}, all of which are provided in full in Appendix A. All metrics are generated over 1000 different randomly generated Moore machines with the input alphabet $I = \langle x, y \rangle$ for each data point as will be further explained below.

Implementation. The algorithm was implemented in Python 3.10 [17] in the AALpy framework of Muškardin et al. [28]. All performance metrics and learned examples as well as the use case in Section 6.2 were conducted with this implementation of the algorithm [39].

6.1.1 Benchmarking Setup

All benchmarks were performed over 1000 randomly generated examples with the input alphabet $I = \langle x, y \rangle$ for each data point provided in this Section. An example in our case is a randomly generated Moore machine that we consider as a black-box system. We then relearn the generated Moore machine and compare the learned model to the originally generated one to determine its correctness, i.e., if we relearned an equivalent version of the original Moore machine. We generated 1000 such examples for every combination of states and the corresponding amount of unique outputs, between 3 and 15 states. This produced over 75 000 different examples, which were used for testing a variety of combinations between number of states, unique outputs and len_{max} values, resulting in over 150 000 tests performed in total.

To better compare the different tables in Appendix A, the same 1000 example were used for the same number of states and unique outputs, i.e., the examples used in Table A.1 with $len_{max} = 10$ are the same ones used for the corresponding rows in Table A.3 with 8 states.

Tables A.1, A.2 and A.3 were all generated using metrics from the performance of the base algorithm from Section 3. In contrast, for sake of comparison, Table A.4 was generated using Angluin's L^* *using resets* algorithm [10] using the existing implementation in AALpy [28]. Each row of every table is the accumulation of runs over 1000 examples as explained above.

The columns in Tables A.1 to A.4 have the following meaning:

- states - the number of states of the generated Moore machines
- unique - the number of unique outputs
- len_{max} - the value of the len_{max} parameter for the 1000 runs
- SUCCESS - the number of successfully and correctly learned models (from 1000)
- TERM - the number of runs (from 1000) that terminated via the no-change condition in Line 7 in Algorithm 1
- LIMIT - the number of runs (from 1000) that terminated because of the limitations described in Section 3.1 via Line 7 in Algorithm 11
- LONG - the number of runs (from 1000) that took longer than 10 seconds to learn or performed more than 100 learning rounds and were aborted to allow for faster testing
- μ-steps - the mean number of steps during all successfully learned runs

- η-steps - the median number of steps during all successfully learned runs
- σ-steps - the standard deviation over all steps during successfully learned runs

A step for our algorithm is a single input and its response performed by the output oracle. The metrics μ-steps, η-steps and σ-steps are calculated for only the successfully learned runs, with at maximum all 1000 if all examples were successfully learned. They do include steps taken during the equivalence queries, however, these have little impact on the metrics because of the usage of a perfect equivalence oracle as described below.

We decided to add the median in addition to the mean because our algorithm tends to have large outliers regarding the number of steps, which can become apparent by comparing the mean to the median or by observing the large standard deviation. Of additional interest are the different types of termination during the algorithm. We will discuss this in more detail below, but generally, TERM is strongly correlated with len_{max} compared to the number of states, while LIMIT is correlated with the number of unique outputs.

Equivalence Oracle. For all benchmarks in Appendix A we used a *perfect* equivalence oracle. A perfect equivalence oracle is one that has access to the underlying Moore machine inside the, normally, black-box SUL and may simply compare the provided hypothesis with the underlying system. It can generate a counterexample simply by determining an input sequence that distinguishes any two states in the automaton and perform the exact sequence afterwards to induce the unexpected behavior in our algorithm. This allows us not to influence the metrics via our choice of equivalence oracle during benchmarking and focus solely on the steps taken during the actual learning process, and not during the equivalence query.

6.1.2 Influence of len_{max}

One factor that plays a big role in the runtime and the successful learning of the algorithm is the len_{max} parameter. It determines the maximum length of words generated from the trace in Algorithm 6 (`findAllWords`) to navigate to target states using Algorithm 4 (`navigateToTarget`). If the direct navigation to a target state does not succeed, words are taken from the trace that led us to states with the correct output before. In cases were we choose a too small len_{max}, if words from the trace are not sufficient to lead us to the target state, we will abandon the target state. This can result in a termination of first Algorithm 11 (`distinguish`) and in consequently, if we should reach the state but cannot, we might terminate the entire algorithm via the alternative condition in Line 7 in Algorithm 1 (`mainAlgorithm`), which is described in Section 3.4. The smaller len_{max} the more likely this scenario is.

Figure 6.1 shows the success rate of learning Moore machines with 8 states, of which a certain number have unique outputs, given a len_{max}. The learning data was aggregated by measuring metrics on 1000 fully-connected, randomly generated Moore machines and observing the number of correctly learned models. Fixating the number of states to exactly 8 allows us to better show the correlation between len_{max} and said number of states of the model. The entire data set for this figure is provided in Table A.1.

As expected, a lower len_{max} leads to the scenario described above more often, which in turn means fewer models will be learned correctly. Further, the combination few unique outputs and lower len_{max} is especially bad in terms of success rate. Of course, it is not affected in case of a *trivial* model, i.e., all states have unique outputs, as we do not require navigation via the trace for such cases. The success rate increases with len_{max} and levels out at some point but does not reach 100% as there are other factors that may lead to a termination. In Table A.1 the TERM value indicates the number of models that were terminated according to the condition in Line 7 in Algorithm 1. LIMIT indicates how many runs terminated due to the limitations described in Section 3.1 and enforced in Line 7 in Algorithm 11, while LONG are outliers and indicate the number of runs that took longer than 10 seconds to learn and were aborted.

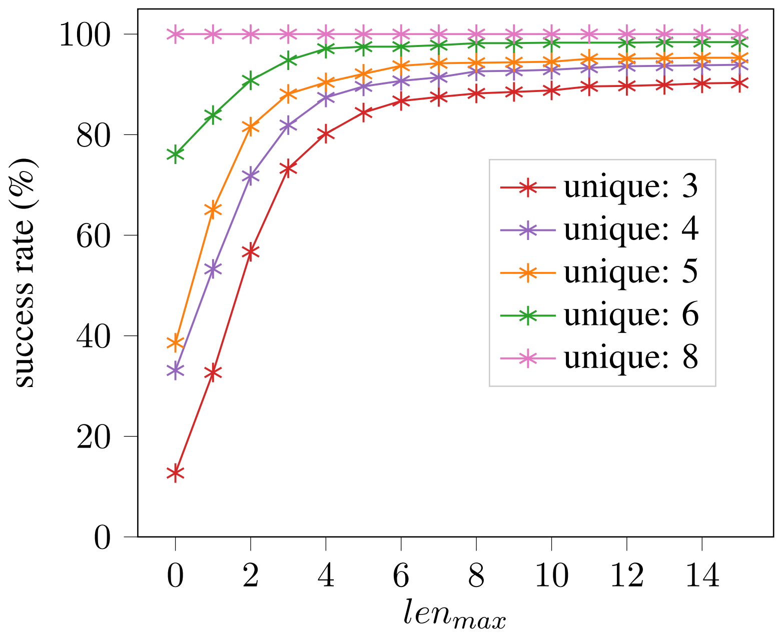

Figure 6.1: Learning success rate depending on len_{max} over 1000 fully-connected, random Moore machines with 8 states from Table A.1.

In short, choosing len_{max} equal to the number of states or higher seems like a good value. In case the algorithm terminates via said condition, len_{max} can always be increased further.

6.1.3 Influence of Unique Outputs

One of the big factors on runtime and the success rate of the algorithm is the number of unique outputs among the states. As we assessed above, len_{max} should be about equal to the number of states to start leveling out success rate. This is exactly what we did in Table A.2 were we adjusted len_{max} to the number of states. The results are visualized in Figures 6.2 to 6.4.

As expected, the number success rate decreases with the number of unique outputs, i.e., 2 unique outputs has the lowest success rate while 12 unique outputs has the highest, as can be seen in Figure 6.2. This makes sense as this is one of the limitations of the algorithm, explained in Section 3.1, which makes 0 unique outputs impossible to learn for the algorithm. But even with a few outputs this limitation can lead to the termination of the algorithm, as not the absolute number of states with unique outputs is required to be greater than zero, but the number of *reachable* states must be. Sometimes during Algorithm 11 it is possible that all other states with unique outputs are unreachable or abandoned, which will trigger the termination via Line 7 even if there would be unique outputs in the machine.

Ideally, the SUL should have as many unique outputs as possible to increase the performance of the algorithm. It works best when only a small percentage of outputs is duplicated.

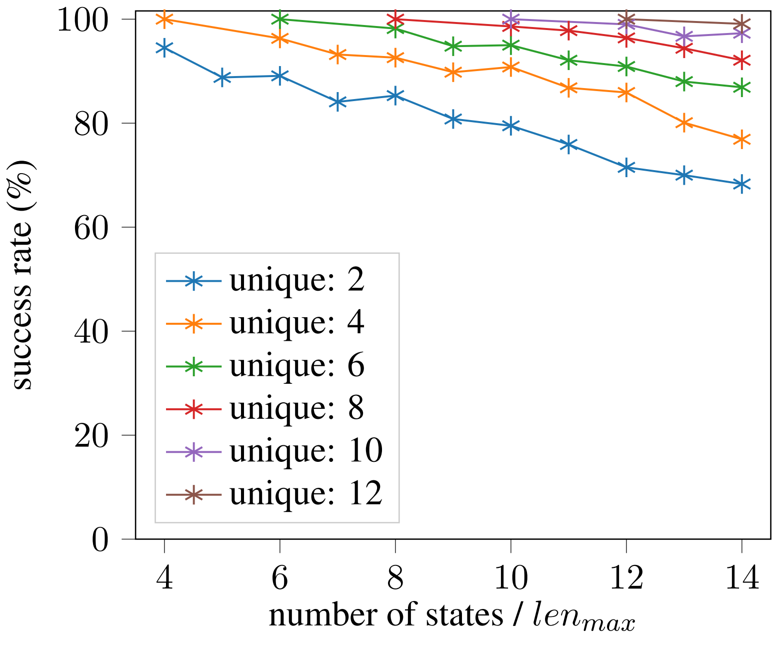

Figure 6.2: Learning success rate depending on the number of states = len_{max} and unique outputs over 1000 fully-connected, random Moore machines from Table A.2.

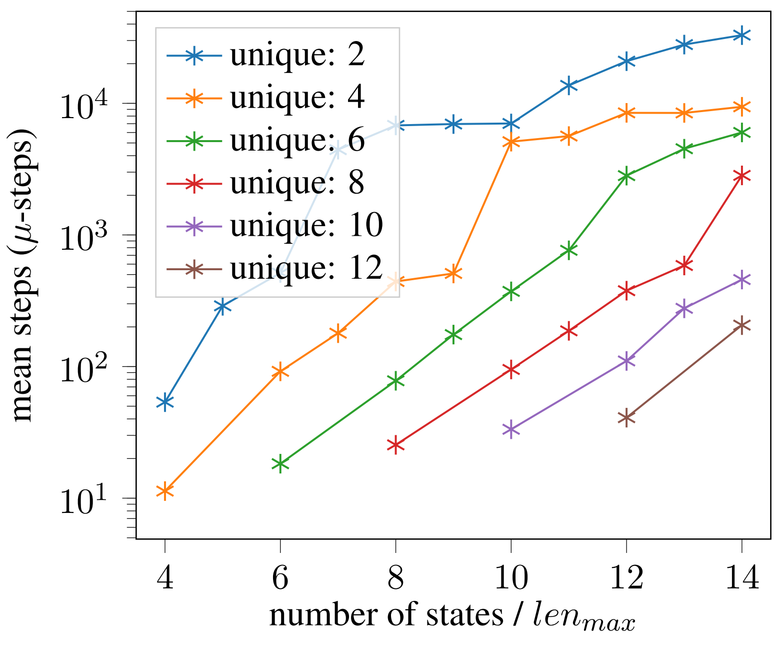

Figure 6.3: Mean of steps during learning depending on the number of states = len_{max} and unique outputs over 1000 fully-connected, random Moore machines from Table A.2.

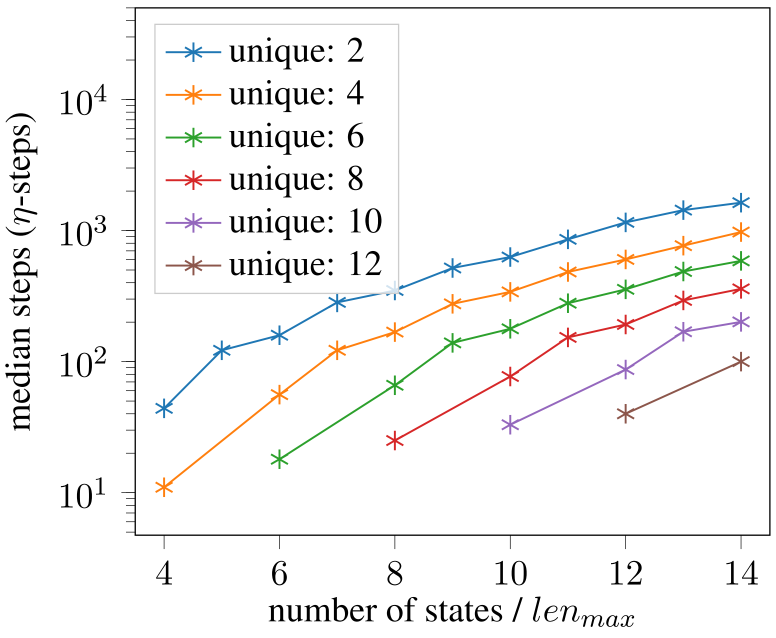

Figure 6.4: Median of steps during learning depending on the number of states $= len_{max}$ and unique outputs over 1000 fully-connected, random Moore machines from Table A.2.

6.1.4 Fixed len_{max}

More often during black-box testing, we have no prior knowledge about the SUL and its states. If the number of states is not known in advance, len_{max} is mostly chosen as some fixed value. This of course has negative repercussions in form of success rate and performance decrease if the number of states turns out to be larger than the chosen len_{max}. However, this is the more realistic example during black-box testing.

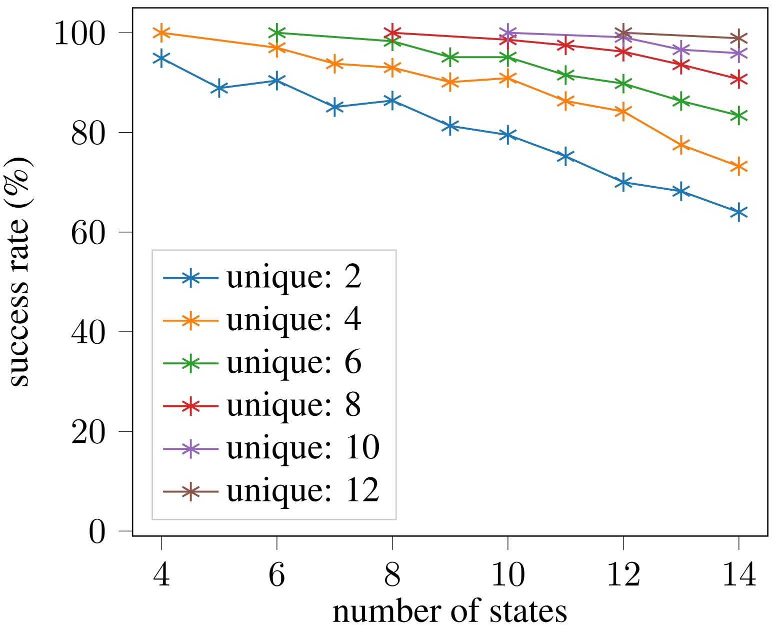

Figure 6.5: Learning success rate depending on the number of states and unique outputs over 1000 fully-connected, random Moore machines with fixed $len_{max} = 10$ from Table A.3.

We provide such metrics with a fixed $len_{max} = 10$ in Table A.3, which we visualize in Figures 6.5 to 6.7. While the changes compared to Table A.2 are not that large, there is a notable trend with better success rate for smaller number of states, as len_{max} is comparatively higher, and a lower one for larger number of states, i.e., more than 10 states.

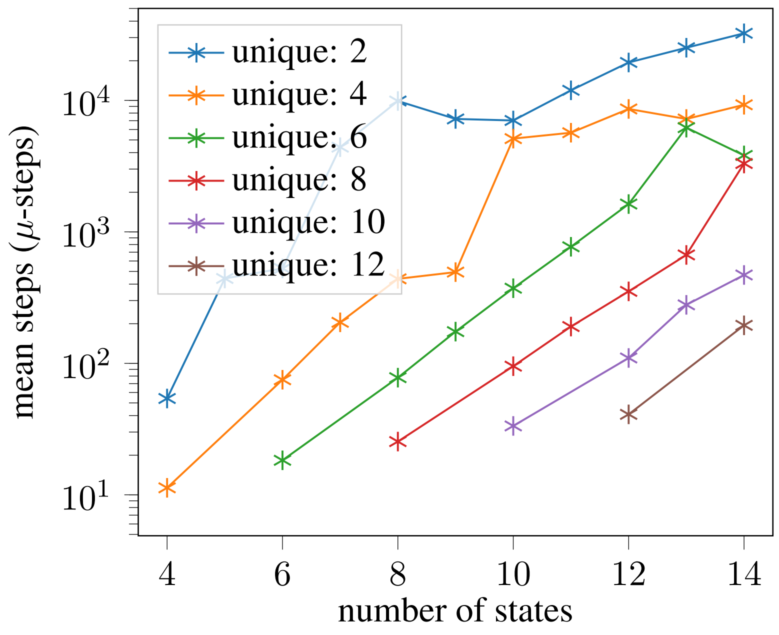

Figure 6.6: Mean of steps during learning depending on the number of states and unique outputs over 1000 fully-connected, random Moore machines with fixed $len_{max} = 10$ from Table A.3.

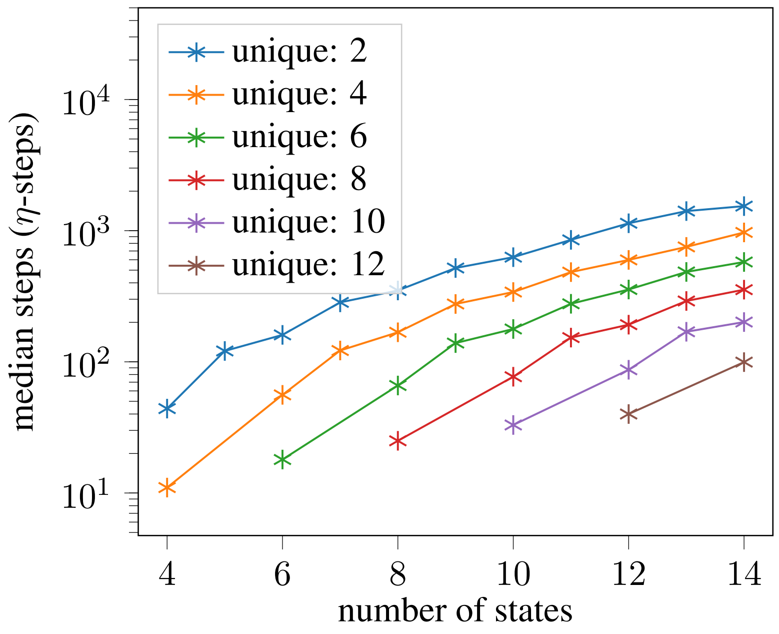

Figure 6.7: Median of steps during learning depending on the number of states and unique outputs over 1000 fully-connected, random Moore machines with fixed $len_{max} = 10$ from Table A.3.

6.1.5 Comparison with L^*

The comparison between L^* and the base algorithm should be taken with a grain of salt, as L^* is allowed to use resets while our base algorithm does not. Additionally, one should keep in mind that fewer runs were successful for the base algorithm than for L^*, especially for lower numbers of unique outputs, and that these do not appear in the metrics about the number of steps. We show the comparison between performance metrics for 8 states in Table 6.1, which were taken from Tables A.2 and A.4.

L^* is much more consistent and is hardly influenced by the number of unique outputs, while the base algorithm is strongly influenced by them, as shown above. That said, even though we do not use resets, our algorithm performs better for trivial examples and for examples with many unique outputs, in this case, 8 and 6 unique outputs. On the other hand, the number of required steps explodes with a small number of unique outputs. However, comparing the medians instead of the means, i.e., ignoring the outliers, we achieve a much more satisfactory performance.

Table 6.1: Comparison between selected data points of L^* and base algorithm performance, taken from Tables A.2 and A.4.

Examples		Learning L^* *with resets*			Learning Base Algorithm		
states/len_{max}	unique	μ-steps	η-steps	σ-steps	μ-steps	η-steps	σ-steps
8	8	129.9	126.0	10.4	25.4	25.0	3.3
8	6	129.9	126.0	11.2	78.0	66.0	40.9
8	5	131.0	126.0	15.4	152.9	133.0	111.9
8	4	131.8	126.0	16.0	442.8	168.0	5695.7
8	3	135.6	130.0	21.6	1747.2	281.5	28588.2
8	2	137.5	130.0	26.1	6796.0	349.0	73021.9

6.1.6 Outliers

One of the weak spots of our algorithm are the outliers in terms of performance. Although all 1000 examples have the same number of states, unique outputs and len_{max} there are a few examples that take a disproportionally large amount of steps and time to complete. This appears to be due to the exact form of the example, i.e., the transitions and order of duplicated inputs, which is hard to quantify.

In the future, it would be of great interest, and would increase the utility of the algorithm, if the outliers could be reduced and therefore the performance would become more stable and predictable. We discuss this further in Section 7.4.

6.2 Case Study: Smoke Meter

6.2.1 System Under Learning: the Smoke Meter

The *AVL 415SE Smoke Meter* device, which we will call Smoke Meter from now on, is an automotive measurement device designed for measuring the soot content of exhaust fumes of combustion engines [11]. It does so by sucking the exhaust fumes through a strip of paper, which is stained by the soot, and measuring the change in reflectivity of the paper. Depending on the strength and length of the suction process, this allows for the analysis of the exhaust fumes and in turn the workings of the combustion engine.

AK-Protocol. The Smoke Meter can be controlled remotely using the AK protocol, a simple Ethernet based communication protocol presented by Jogun [23]. An interaction with the Smoke Meter is always initialized by the oracle by sending an AK telegram to the Smoke Meter, to which it immediately responds with its own AK telegram. This interaction is only limited by the speed of the Ethernet connection and the tiny delay the Smoke Meter requires to process the command, which in our setup amounted to about 0.12 seconds on average. Each AK telegram contains a four letter AK command followed by potentially other values depending on the type of command sent. AK commands starting with the letter `S` perform actions on the device and may lead to state transitions, which we call `S`-commands, while ones starting with `A` only query device parameters and never result in state changes, called `A`-commands.

Input Alphabet. The input alphabet, with which the device was learned, consists of a range of different `S`-commands, as we are interested in observing state transitions, which are only performed on `S`-commands. It does not include all possible `S`-commands but a selected subset, which we constructed jointly with AVL, based on what we believed to be most interesting or are composed of commands that we believe to be used more often during ordinary device usage. The only exception is the `WAIT` input, which is not an AK command but is instead the waiting input introduced in Section 5.1. The `WAIT` command is not sent to the Smoke Meter, as it starts neither with `S` nor `A`, but is processed by the output oracle.

- `SRDY` - Stops all current activities and sets the device to Standby mode.
- `SMES` - Starts a measurement cycle, sucking in exhaust fumes and measuring the amount of soot on the paper.
- `SPUL` - Starts a purging cycle, cleaning the air ducts.
- `SFPF` - Starts a paper feed.
- `SVOP` - Starts a volume check, testing the air flow through the device.
- `SMKA` - Starts a measurement head calibration.
- `SMRE` - Starts a reflection measurement on the current paper segment.
- `SLEC` - Starts a leakage check.
- `SASB` - Prepares the device for a synchronized measurement by readying it for sucking on the next `SMES` command.
- `SBSZ` - Reset operating hours of device.
- `SKOR` - Calibrate volume correction factor.
- `WAIT` - The waiting input, which is not an AK command and instead performs the waiting action described in Section 5.1.

Internal State. The Smoke Meter responds to any `S`-command immediately with a payload containing the information about the success of the operation, in the form of either a code *Ok* or *Error*[1]. This response is mostly not useful for determining the state of the device, as all `S`-commands return the same *Ok* and *Error* codes independent of the states they are in. To clarify, there might be a difference between states if one operation returns *Ok* in one state and *Error* in another, but this information is not enough to differentiate between most states. For example, the success response *Ok* is identical for all `S`-commands in all states, which does not provide any relevant state information.

Of much greater interest are the internal state variables that can be accessed with `A`-commands. These return a wide variety of sensor data or internal data points about the current state of the device. Additionally, `A`-commands do not change the state of the SUL and may be performed at any time, no matter which state the device is in or which process is currently executed on the device.

[1]This is a simplified explanation. The response to `S`-commands is actually either a zero for success or the number of a specific error code.

Similar to how Aichernig et al. [3] determined the state for their industrial measurement devices, we will use selected A-commands to query the state of the device. We will interleave these and the S-commands to first change the state of the SUL, using an S-command, directly followed by querying the device state, using an A-command. The response to any input, i.e., S-command, will then be the response of the subsequently performed A-command instead, which provides the state information.

For this use case, we chose the AIZU command. More specifically, the AIZU command contains a number of different values in its response, one of which is the *internal state* of the device as described in the manual of the Smoke Meter [11]. This internal state variable is a number which gives information about what the device is currently doing. However, the state information provided by AIZU is not extensive enough to provide unique outputs for every single state code, i.e., the values of the internal state variable are too coarse to differentiate between some of the states for a state machine, which is the reason learning is required in the first place.

The potential state codes provided by the AIZU command are:

- 0: power up - Before the device is fully turned on.
- 1: error - An error occurred which should be acknowledged using the SRDY command.
- 2: set ready - Stopping operations and preparing for return to Standby mode.
- 3: ready - Standby mode, in which the device is ready for measurements or other commands.
- 4: ready with purge - Purging the air vents and preparing for return to Standby mode.
- 5: device check - Performing some device performance or calibration check.
- 6: volume check - Testing the air flow through the device.
- 7: paper measurement - Measuring the reflection of the paper and calibrating the device.
- 8: measurement - Performing a standard measurement.
- 9: set suction ready - Preparing for synchronized measurement.
- 10: suction ready - Ready for synchronized measurement.
- 11: synchronized measurement - Performing a synchronized measurement.
- 13: paper feed - Advancing the paper feed.

These internal state codes are used at the output to the states of our Moore machine for the Smoke Meter. We assume that not observing a difference in the response to a query, i.e., sending an S-command followed by AIZU, means that no state change took place, which results in a self transition on that input. Furthermore, the polling of the state in response to the waiting input WAIT, explained in Section 5.1 and visualized in Figure 5.4, is performed using the AIZU command, i.e., the empty input ϵ is replaced by AIZU, which returns the current state.

Equivalence Oracle. For equivalence queries we used a modified random walk oracle. A classical random walk oracle performs up to $step_{max}$ random inputs from the input alphabet and has a certain probability of performing a reset, p_{reset}, after every step. Additionally, our modified version of the random walk oracle was aware about the definition of timed states in Section 5.2 and only performed waiting inputs in such states while still choosing random inputs in all others. This prevents us from performing any missing inputs in a timed state of the partial hypothesis provided to the equivalence oracle, which was explained in detail in Section 5.2.

6.2.2 System Under Learning: the Digital Twin

In addition to learning the physical Smoke Meter device we also learned a model from a *digital twin* (DT) of the Smoke Meter. A DT is a virtual model, i.e., a simulation, that represents and behaves as a physical device, which is one of the definitions provided by Preuveneers et al. [33]. Additionally, Tao et al. [36] explain common uses and applications of DTs in industry as well as show different views on the concept of DTs. One of the goals of the LearnTwins Project is to improve said DTs through the use of automata learning.

The DT of the Smoke Meter is used by the company AVL in-house for testing, among others, and was at some point manually created based on the product guide of the Smoke Meter. It responds to AK telegrams and can be interfaced with exactly like the real Smoke Meter in Section 6.2.1. Under ideal circumstances, the DT should not be differentiable from the real device, except that the waiting times defined for the DT were much shorter than the once on the real device to facilitate faster testing. As will be shown in our results in Section 6.2.3, we managed to find further subtle differences in the behavior between the real device and the DT, which were then used to enhance and improve the existing DT of AVL.

As the DT mirrors the behavior of the Smoke Meter, can be interacted with the AK protocol and otherwise provides the same responses, all commands and settings explained for the real Smoke Meter in Section 6.2.1 are also applicable for the DT.

6.2.3 Results

The learned model of the Smoke Meter, explained in Section 6.2.1, and the provided DT, explained in Section 6.2.2, are shown in Figure 6.8 and 6.9 respectively.

We enhanced both models with timing information from the learning process. The `WAIT` transition on timed states has three additional values in parenthesis next to it, namely m, d and N, where N is the number of times this transition was taken during the learning process, while m and d are the mean and the standard deviation of times waited on this transition respectively. For example, if a transition was taken three times, $N = 3$, while waiting 3, 5 and 7 seconds respectively, we would label the transition with a mean $m = 5$ and a standard deviation $d = 2$.

Learning Parameters. For the learning of the models we chose the following parameters:

- $step_{max} = 150$ - the maximum number of steps for the equivalence oracle
- $p_{reset} = 0$ - the reset probability for the equivalence oracle
- $i_w =$ `WAIT` - the waiting input
- $t_w = 200$ - the maximum waiting time in seconds
- $t_p = 0.2$ - the polling interval in seconds
- $len_{max} = 10$ - the maximum word length generated from the trace
- I - the input alphabet provided in Section 6.2.1
- Finally, the internal state codes provided by the `AIZU` command are used as the outputs of our Moore states as described in Section 6.2.1.

The equivalence queries took a disproportionally long time, primarily because most steps taken were waiting actions, which of course could take up to $t_w = 200$ seconds. We chose only a small $step_{max}$ to keep the time of our equivalence queries short. Similarly, we decided to reduce the reset probability, p_{reset}, to 0%, as navigating the reset sequence would take additional time for every reset performed and did otherwise not make a big difference for these fully-connected models, especially with a random walk.

The maximum waiting time t_w had to be chosen so high because some of the transitions in the Smoke Meter took a comparatively long time. Especially, the transition from State 16 to State 4 took two minutes on average. We did perform one single learning round with a waiting time of 5 minutes before choosing $t_w = 200$ to make sure that no longer transitions existed in most of the explored states.

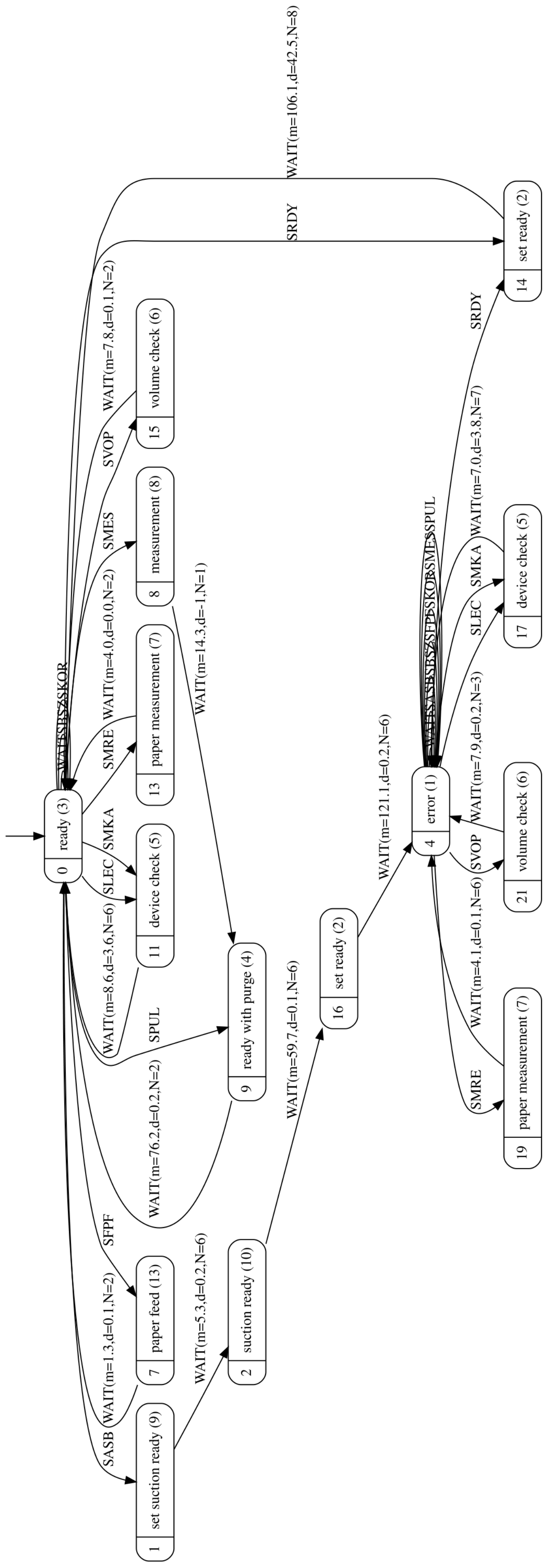

Figure 6.8: Partial learned model of the AVL Smoke Meter measurement device with timings.

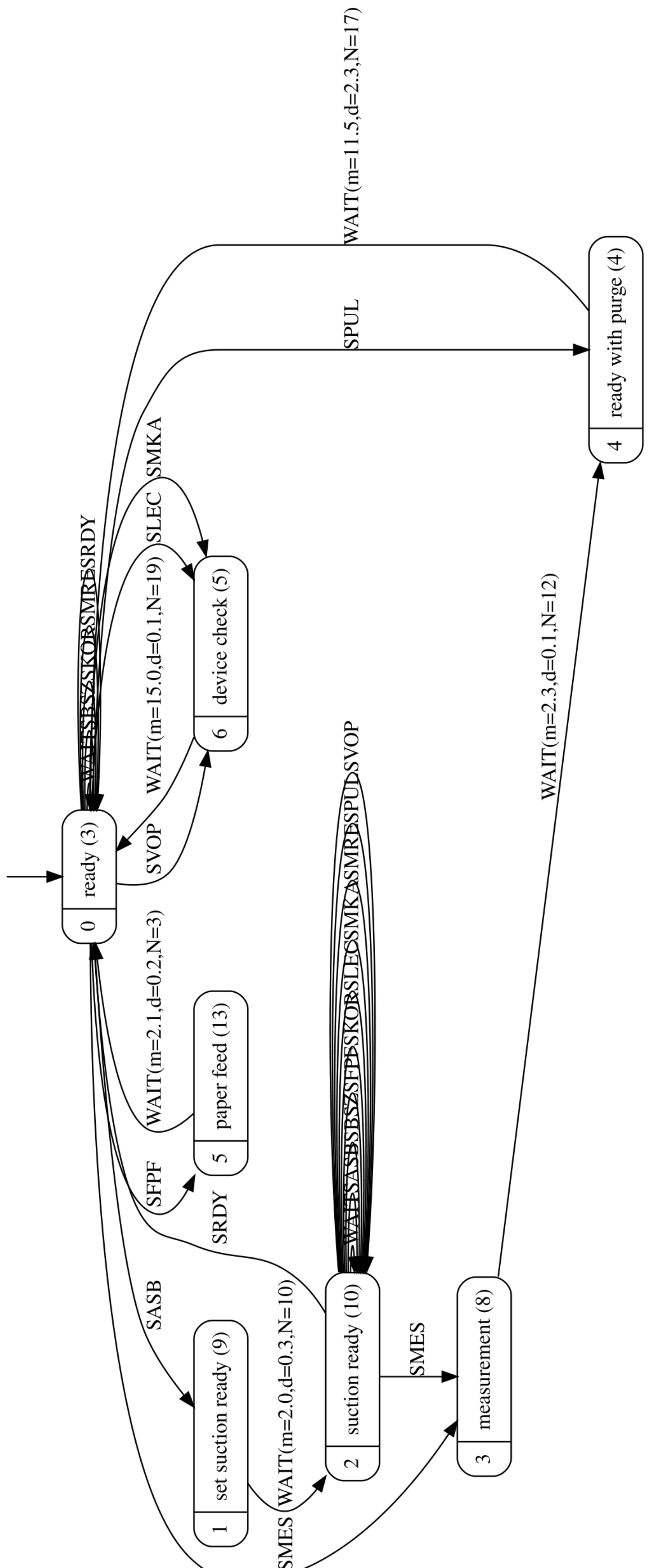

Figure 6.9: Partial learned model of the digital twin of the AVL Smoke Meter measurement device with timings.

Finally, the polling time t_p was picked as small as possible to get more accurate waiting times for the models.

Differences. As can be seen from comparing Figures 6.8 and 6.9, some differences can be observed, the biggest being that the state *suction ready (10)* is a timed state for the Smoke Meter in Figure 6.8, which in turn leads to the state *error (1)*, while it is not timed for the DT in Figure 6.9. This behavior was not documented in the product guide and, because the DT was created by hand from descriptions from the product guide, was therefore not included in the DT. Without this additional error state, the DT turned out to be *trivial*, i.e., all of its outputs are unique, which resulted in only a single learning round for exploring the DT.

Another difference is that some commands like `SVOP` do not lead to the correct state in Figure 6.9, in this case it leads to *device check (5)* instead of *volume check (6)*, while other commands do not lead to state transitions at all, such as `SMRE`.

Lastly, as already mentioned in Section 6.2.2, the DT does wait for a significantly shorter time on transitions than the real device, which was done by design to improve tests that did not require exact times on the waiting transitions.

Implications. Altogether, the results obtained from learning the models were satisfactory and presented some opportunities for us to enhance and improve the DT. The missing or incorrect state transitions, as explained above, were added to the DT. In addition to that a new timing profile was created based on the timings obtained from the real device in Figure 6.8. This profile can be exchanged for the one with shorter waiting times in the DT whenever there is a special interest in simulating the device as close as possible, in comparison to performing the tests faster. Additionally, the learned models were validated by AVL and seem to be correct representations of the device.

7 Concluding Remarks

7.1 Summary

In this thesis we presented an active automata learning algorithm which learns Moore machines resetless. We extended the base algorithm to handle time-triggered state transitions in order to learn industrial measurement devices that exhibit such time-triggered behavior.

The algorithm first engages in state exploration, in which all missing inputs are tried and processed. It then calculates distinguishers for the W_o-sets, which are then split up into output specific classical W-sets used by the algorithm, by finding inconsistencies in the trace. These can then be used to distinguish states with the same output until our hypothesis is deterministic, which will then be given to the equivalence oracle to ensure its correctness. These steps are then repeated as necessary.

We provide the formal data structures, methods and analysis of both in great detail. In addition, we demonstrate the workings of the algorithm on two illustrative examples, presenting every step the algorithm performs in-depth.

We explain time-triggered behavior, that it introduces non-deterministic behavior into the algorithm, and that we can extend our base algorithm to deal with it by defining a special waiting input. It is used to wait for timed transitions while polling the current state to detect sudden state changes. Where we encounter time-triggered behavior we only explore said behavior and learn a partial model instead, which prevents us from encountering said non-deterministic behavior during the exploration of other inputs in timed states. The instructions to extend the base algorithm are provided in full.

We implemented the base algorithm and the extension in Python [17] using the AALpy framework [28] and provided it here [39]. We used this implementation to test the success rate and performance of our algorithm on over 75 000 different randomly generated Moore machines with a variety in number of states and unique outputs. Furthermore, we provide an in-depth analysis about these metrics depending on the size of the automaton, the number of unique outputs and the len_{max} parameter. The algorithm learns well for small to medium sized automata and works better when the number of unique outputs is higher. On the other hand, while most automata can be learned in normal time there are a considerable amount of outliers that make the algorithm overall less consistent than desirable.

Finally, we performed a case study on our algorithm used on an industrial measurement device. We were able to successfully learn the Smoke Meter, an automotive industrial measurement device by AVL, and its digital twin and found differences in their behavior. These results were then used to improve the digital twin and extend it with more precise timing information.

7.2 Related Work

In the domain of automata learning, we differentiate between passive and active learning approaches. Passive automata learning is the learning of a model from existing traces, without interacting with the SUL at all. Examples for prominent passive learning algorithms, as mentioned by the survey of Aichernig et al. [4], are the RPNI [30] and ALERGIA [14] algorithms, both using a state-merging method to construct the models from the traces.

In contrast to this is the active automata learning approach, which interacts with the SUL via queries to learn the model. Among the most prominent active learning algorithms are Angluin's L^* [10], Isberner et al.'s TTT [21], as well as Kearns and Vazirani's algorithm [24], all of which were evaluated in Aichernig et al. [8]. Both AALpy [28] and LearnLib [22] are libraries that support a variety of the aforementioned algorithms.

First Reset-less Learning Approach. The first procedure that allowed for reset-less active automata learning was described by Rivest and Schapire [34]. The key technique they use for it is the inference of a *homing sequence* (see Section 2.5). They use the L^* algorithm introduced by Angluin [10] but replace the required reset with said homing sequence instead. Because a homing sequence is not guaranteed to end in a particular state, just a known one, they simulate up to n different versions of L^*, one for each state after a homing sequence. L^* learns the SUL correctly if the homing sequence is indeed correct, however, the homing sequence is not known in advance and must be constructed during the learning of the algorithm. The homing sequence will be improved every time some inconsistency is detected in L^*, in which case a more general homing sequence is generated and then the L^* procedure may be started again with this better homing sequence.

Reset-less Passive Learning. Chen et al. [15] present an approach on passively learning Markov chains from a single trace of observations. Their approach only requires a single trace because their focus lies on learning devices for which it is difficult to obtain multiple independent execution traces. Such a trace may be generated without any resets and thus presents a viable method for passive, reset-less learning.

hW-inference Algorithm. Another modern approach to reset-less learning is the newly developed *hW-inference* algorithm proposed by Groz et al. [19], which can learn Mealy machines by assuming a bound on the number of states. The hW-inference works by constructing a *characterization set* as well as a *homing sequence* and gradually refining both until all states can be differentiated. The main algorithm assumes a perfect homing sequence and W-set. Using these, a hypothesis can be built by progressively exploring the SUL. If the homing sequence and W-set are indeed perfect, then no non-determinism will occur and no counterexample is required to refine the hypothesis. If non-determinism does occur, then either the homing sequence or the W-set or both are not yet correct. They use a number of methods and heuristics to then improve the homing sequence or the W-set respectively and repeat this process as long as necessary until the main algorithm terminates. Bremond and Groz [13] assess the performance of the hW-inference algorithm on a number of case studies.

Our own algorithm is somewhat similar to hW-inference in some aspects but differentiates itself through the following points:

Firstly, our algorithm works specifically on Moore machines. We exploit the structure of said Moore machines multiple times. Once during the polling of the state information to deal with time-triggered behavior, as explained in Section 5.1. Another time we benefit from the form of Moore machines, as it allows us to split the classical W-set into multiple smaller W_o-sets, leading to fewer distinguishers per state overall, as described in Section 3.3.5.

Additionally, our algorithm does not use homing sequences for navigating the SUL, but generates potential transfer sequences from the trace instead in Algorithm 6. We also use the trace to generate distinguishers for our W_o-sets in Algorithm 10.

Learning Industrial Measurement Devices. Finally, others have worked with the types of measurement devices, which we aimed to learn with our algorithm, before:

Aichernig et al. [3] are among the most closely related one. They learned a different device but with the same AK protocol and, in their work, encountered sporadic non-determinism from the device, which they were able to mask using sink-states.

Additionally, Aichernig et al. [1] also tested the same device using a model-based mutation testing approach. In said approach they check for conformance between a test model of the systems and generated mutations, i.e., changes induced in the model, to potentially detect discrepancies and bugs. However, for this a model must first be created, potentially using model learning techniques, which is where our algorithm could be used.

Also, Aichernig and Burghard [2] demonstrate a mapping from domain-specific models to Moore machines for use in model-based testing. They work on similar industrial devices and use a domain-specific language called the Measurement Device Modeling Language, which they use to transform the previously mentioned models into partial Moore machines.

7.3 Conclusion

The research problem of this thesis was to develop a new active automata learning algorithm that works without resets and models the systems as Moore machines. The developed algorithm was then adapted to specifically learn the models of measurement devices from the automotive industry such as described by Aichernig et al. [3], by building an extension for the algorithm that allowed it to handle time-triggered behavior during execution. We model these devices as Moore machines because they represent their capabilities better. They have some internal state variables and sensor values which we can access using queries. This corresponds to the output of a Moore state. We exploit the structure of Moore machines in our algorithm to perform fewer input sequences to distinguish states, by splitting the classical W-set into output specific W_o-sets, explained in Section 3.3.5.

Resetting the learned system is sometimes difficult or infeasible due to constraints or would otherwise require an untenable amount of time. Therefore, the reset-less nature of the algorithm was an especially important aspect of the development process. The developed algorithm does in fact not require resets at all, but is still able to simulate resets for testing algorithms during equivalence checking by navigating to a chosen starting state. This allows us to use any established testing algorithm, such as the W-method by Chow [16], the mutation testing approach by Aichernig and Tappler [7] or other ones evaluated in Aichernig et al. [8], in combination with the base form of our learning algorithm.

Using the extension to deal with time-triggered behavior requires some changes to the exploration and testing aspects of the algorithm. In the extension we define a waiting input that represents the waiting on a timed state transition for a certain amount of time. Once a time-triggered state transition is recorded no other inputs are tried in that state, which prevents the occurrence of non-determinism due to race conditions. Otherwise, these race conditions could lead us to observing transitions incorrectly, which would prevent the base algorithm from learning devices with such behavior, e.g., the Smoke Meter [11]. Using the extension, we learn partial models instead, as only the waiting input is performed in states with time-triggered behavior.

We evaluated our base algorithm on a number of different combinations of amount of states, unique outputs and the len_{max} parameter. We did this on over 75 000 randomly generated models, performing each data point on 1000 of them to increase the validity of our results. Our evaluation shows that our algorithm learns small to medium sized automata well for a high number of unique outputs. Furthermore, it shows that the success rate and performance of the algorithm is strongly correlated to the number of unique outputs in the SUL. The fewer unique outputs the lower the success rate and the worse the performance and vice versa. In fact, one of the limitations of our algorithm is that it requires at least one reachable state with a unique output to distinguish between states. If this requirement is not fulfilled the algorithm terminates without learning the model. On the other hand, trivial systems, i.e., SULs with only unique outputs, can be learned with a 100% success rate while outperforming Angluin's L^* algorithm [10] with resets.

Another factor in success rate and performance is the len_{max} parameter. It determines the maximum length of words extracted from the trace that we use for navigating the SUL. A larger len_{max} generally means a higher success rate, but also more required steps to learn the model and vice versa. We determined that len_{max} equal to the number of states of the SUL generally gives quite good results, but increasing it further is the safest option.

Finally, we also present our case study about the Smoke Meter [11]. We successfully learned the device in a practical setting. Additionally, we learned the model of an existing DT of it. We managed to find differences in behavior between the two devices, showing that our learned model of the real device was more thorough than the implementation of the DT. The learned model of the Smoke Meter device was ultimately used to improve the DT and outfit it with more precise timing information about the time-triggered transitions.

In summary, we developed a reset-less active automata learning algorithm that works on Moore machines. Our algorithm can learn most Moore machines depending on the number of unique outputs of the SUL. There are multiple ways in which our existing algorithm can be improved, chief among them the removal of the limitation and the dependency on the number of unique outputs. Finally, we can say that our algorithm fulfills the given requirements and accomplishes the task of learning the industrial measurement device, the Smoke Meter, successfully.

7.4 Future Work

There are broadly three different directions in which future work could lean: Performing an extended evaluation or adding additional case studies, improvements to success rate and performance, or adapting the algorithm to work with a different class of automata.

Extended Evaluation. We did present our evaluation in Chapter 6 where we presented the results of performing the algorithm on over 75 000 randomly generated Moore machines and a case study with an industrial measurement device. However, extending the evaluation or presenting new scenarios in which to analyze the success rate and performance of the algorithm would be potential tasks for the future, as this would allow for more insight into the possible applications of the algorithm.

Our algorithm could be tested with a variety of different testing algorithms, such as the benchmarking presented by Aichernig et al. [8]. In our own evaluation in Section 6.1 we were only interested in the performance of our learning algorithm and chose a perfect equivalence oracle for our tests, which does not exist for real black-box systems. Therefore, a performance analysis about combining our learning algorithm with different testing ones could be a good next step in establishing our algorithm.

We presented our case study about the Smoke Meter [11] device in Section 6.2. While we did manage to successfully learn the device, we only presented a single real world example. Additional case studies on other devices could help establish if our algorithm is suitable for learning practical examples in the field, which is already planned in the LearnTwins project.

Improvement of Existing Algorithm. In the evaluation in Chapter 6 we analyze the success rate and performance of our algorithm. While the algorithm performs adequately for the majority of examples there are some outliers that make the algorithm less consistent than desirable. Some improvements regarding the success rate, performance or general behavior of the algorithm could make it more viable overall.

At the moment, the algorithm is very strongly tied to the number of unique outputs of the underlying system. Not only is there a hard requirement that there exists at least one unique output, as stated in Section 3.1, but the performance is also strongly coupled to it, with a much lower success rate and increased runtime for systems with few unique outputs. One of the major future goals would be to ideally remove this limitation entirely or to at least mitigate the effects it has on the runtime and success rate. Improving this would allow the algorithm to be used in a more general setting and make it more viable and consistent for usage with a wider range of devices.

Our algorithm calculates the distinguishers for our W_o-sets in Algorithm 10 by extracting the observed differences from the trace. At the moment, all such observed sequences are actually used as distinguishers, with the exception of ones that are already prefixes of existing distinguishers as presented in the optimizations in Section 3.9. This may result in large W_o-sets, which in turn increase the algorithm's runtime as more distinguishers have to be tried in Algorithm 11 (`distinguish`). Rivest and Schapire [34] introduce a method that allows them to efficiently find sequences to distinguish states from received counterexamples. They use their method to improve upon the L^* algorithm. Their approach could potentially be useful for more efficiently selecting distinguishers in Algorithm 10 and thus reducing the size of the W_o-sets and in turn the runtime of the algorithm.

One of the downsides of our approach in Chapter 5 is that we only learn a partial model and are missing all other outputs for timed states. We do not perform other inputs because we do not want to perform a time-triggered transition during the sending of another input, as this would introduce non-deterministic behavior. We could improve our algorithm by measuring the time until the time-triggered transitions occur in each state and then perform other inputs while we have enough time. This could work, with some changes, within our existing extension from Chapter 5. Another approach would be to explore these other inputs in timed states through fuzzing, either during the learning process or afterwards. Aichernig et al. [5] present a method to perform learning-based fuzzing of black-box systems that could be adapted to work on our algorithm.

Adaption to other Classes of Automata. In addition to the improvements that could be made to the algorithm there are a number of extensions that could transform the algorithm to work on a different class of automata or to allow it to work under more general circumstances.

For example, a potential next step could be to adapt the algorithm to run on Mealy machines, such as the hW-inference by Groz et al. [19] among others. The current methods presented in this paper do take advantage of the Moore machines that are currently used, so it is an open question if the methods from our algorithm can be adapted in such a way without entirely discarding the structure of our algorithm.

Among the most interesting of the possible extensions is the idea of adapting the algorithm to work on timed automata. Tappler et al. [37] propose an approach to learn models from timed traces using real-valued clocks, which is then adapted to an active approach by Aichernig et al. [6]. We currently use the time-triggered extension presented in Chapter 5 to deal with the timed transitions, which leaves us only with partially learned models. Embracing the timed behavior and modeling our underlying systems as timed automata could allow us to achieve fully explored models instead of only partial ones.

Finally, a possible extension includes the handling of non-deterministic behavior. Originally, this was one of the planned requirements for this algorithm but turned out to be outside the scope of this thesis' work. Aichernig et al. [3] learned very similar industrial measurement devices with sporadic non-deterministic behavior by masking the occurring non-determinism with sink-states. Such an extension could be a potential next step by combining this method with our algorithm. Pferscher and Aichernig [32] present another approach that works on non-deterministic automata. By merging akin states they learn a more general model of the black-box system, which also allows them to learn models with large input and output alphabets. This poses another potential way in which our algorithm could be extended in the future.

Appendices

A Learning Metrics

Table A.1: Algorithm learning results over 1000 strongly-connected, random Moore machines with 8 states, of which a certain number have unique outputs, and a given len_{max}.

Examples		Learning Success/Failure				Learning Steps		
unique	len_{max}	SUCCESS	TERM	LIMIT	LONG	μ-steps	η-steps	σ-steps
8	0	1000	0	0	0	25.4	25.0	3.3
6	0	761	191	26	22	68.3	60.0	28.8
5	0	386	490	106	18	102.8	91.0	44.6
4	0	331	562	96	11	132.0	120.0	56.2
3	0	127	714	148	11	178.6	156.0	80.6
8	1	1000	0	0	0	25.4	25.0	3.3
6	1	839	122	20	19	70.8	62.0	30.5
5	1	651	252	72	25	131.8	119.0	80.5
4	1	533	389	69	9	158.7	142.0	106.9
3	1	327	535	122	16	231.6	208.0	120.7
8	2	1000	0	0	0	25.4	25.0	3.3
6	2	908	73	7	12	74.1	64.0	33.2
5	2	816	134	38	12	143.5	127.5	93.1
4	2	718	234	43	5	178.8	158.0	136.4
3	2	567	343	78	12	369.6	243.0	2129.1
8	3	1000	0	0	0	25.4	25.0	3.3
6	3	948	40	7	5	76.8	65.0	42.1
5	3	881	74	36	9	146.7	129.0	94.4
4	3	819	136	42	3	205.8	163.0	357.0
3	3	733	192	69	6	586.0	266.0	3664.3
8	4	1000	0	0	0	25.4	25.0	3.3
6	4	971	20	7	2	79.7	66.0	65.0
5	4	904	54	36	6	149.6	130.0	101.7
4	4	874	84	38	4	423.2	166.0	5839.5
3	4	802	133	59	6	1565.1	273.5	29463.1
8	5	1000	0	0	0	25.4	25.0	3.3
6	5	975	16	7	2	78.4	66.0	45.2
5	5	921	39	36	4	149.0	133.0	95.4
4	5	896	62	38	4	616.9	166.0	7978.7
3	5	844	95	58	3	1686.8	275.5	28901.5
8	6	1000	0	0	0	25.4	25.0	3.3
6	6	975	17	7	1	78.0	66.0	44.4
5	6	937	23	36	4	152.8	135.0	104.6
4	6	907	52	37	4	427.7	167.0	5737.8
3	6	867	75	58	0	1791.6	278.0	28854.5
8	7	1000	0	0	0	25.4	25.0	3.3
6	7	978	15	6	1	77.4	66.0	37.1
5	7	942	19	36	3	155.4	133.5	152.6
4	7	914	43	37	6	427.1	167.0	5715.8
3	7	875	67	58	0	1756.4	281.0	28702.3
8	8	1000	0	0	0	25.4	25.0	3.3

6	8	982	12	6	0	78.0	66.0	40.9
5	8	943	18	36	3	152.9	133.0	111.9
4	8	926	31	37	6	442.8	168.0	5695.7
3	8	882	60	58	0	1747.2	281.5	28588.2
8	9	1000	0	0	0	25.4	25.0	3.3
6	9	982	12	6	0	77.9	66.0	40.8
5	9	944	18	36	2	153.0	133.0	112.0
4	9	927	30	37	6	439.9	168.0	5691.3
3	9	885	57	58	0	1746.6	281.0	28539.9
8	10	1000	0	0	0	25.4	25.0	3.3
6	10	983	11	6	0	77.8	66.0	40.6
5	10	945	17	36	2	153.8	133.0	115.9
4	10	930	28	37	5	439.0	168.0	5682.0
3	10	888	54	58	0	3454.0	283.0	58562.2
8	11	1000	0	0	0	25.4	25.0	3.3
6	11	983	11	6	0	77.8	67.0	40.6
5	11	951	12	36	1	154.2	134.0	113.3
4	11	933	25	37	5	439.1	168.0	5672.9
3	11	896	47	57	0	3437.1	284.5	58300.8
8	12	1000	0	0	0	25.4	25.0	3.3
6	12	983	11	6	0	77.9	67.0	40.7
5	12	951	12	36	1	154.2	134.0	113.4
4	12	936	24	37	3	441.0	168.0	5664.4
3	12	897	46	57	0	3435.6	285.0	58268.3
8	13	1000	0	0	0	25.4	25.0	3.3
6	13	984	10	6	0	78.3	67.0	43.3
5	13	952	12	36	0	154.3	135.0	113.5
4	13	937	23	37	3	436.1	168.0	5657.1
3	13	899	44	57	0	3426.4	285.0	58203.6
8	14	1000	0	0	0	25.4	25.0	3.3
6	14	984	10	6	0	78.6	67.0	46.4
5	14	953	11	36	0	153.9	134.0	112.6
4	14	938	23	37	2	435.4	168.0	5654.1
3	14	902	41	57	0	3416.9	286.0	58106.9
8	15	1000	0	0	0	25.4	25.0	3.3
6	15	984	10	6	0	77.9	67.0	39.7
5	15	953	11	36	0	153.9	134.0	112.6
4	15	939	23	37	1	441.3	168.0	5653.7
3	15	903	41	56	0	3413.6	286.0	58074.7

Table A.2: Algorithm learning results over 1000 strongly-connected, random Moore machines with a given number of states $= len_{max}$, of which a certain number have unique outputs.

Examples		Learning Success/Failure				Learning Steps		
states/len_{max}	unique	SUCCESS	TERM	LIMIT	LONG	μ-steps	η-steps	σ-steps
3	3	1000	0	0	0	8.1	8.0	1.2
4	2	945	19	36	0	53.6	44.0	35.7
4	4	1000	0	0	0	11.3	11.0	1.6
5	2	888	31	80	1	289.4	122.0	2138.5

5	3	958	16	26	0	68.1	48.5	91.3
5	5	1000	0	0	0	14.7	15.0	2.1
6	2	891	52	56	1	514.3	159.0	3444.0
6	3	910	34	53	3	153.1	119.0	182.4
6	4	963	19	17	1	92.0	56.0	656.7
6	6	1000	0	0	0	18.3	18.0	2.4
7	2	841	61	96	2	4447.3	283.0	78616.3
7	3	941	23	32	4	1587.2	161.0	28984.0
7	4	932	26	42	0	179.3	122.0	840.6
7	5	974	17	9	0	89.4	62.5	299.5
7	7	1000	0	0	0	22.0	22.0	2.8
8	2	853	56	79	12	6796.0	349.0	73021.9
8	3	882	60	58	0	1747.2	281.5	28588.2
8	4	926	31	37	6	442.8	168.0	5695.7
8	5	943	18	36	3	152.9	133.0	111.9
8	6	982	12	6	0	78.0	66.0	40.9
8	8	1000	0	0	0	25.4	25.0	3.3
9	2	808	87	86	19	6959.1	518.0	59018.6
9	3	875	65	56	4	3421.2	353.0	62344.4
9	4	898	51	48	3	509.7	276.0	1796.6
9	5	939	22	38	1	291.9	173.0	1162.0
9	6	948	19	28	5	174.9	139.0	238.8
9	7	988	7	5	0	102.7	73.0	298.5
9	9	1000	0	0	0	29.3	29.0	3.9
10	2	795	92	85	28	7029.9	628.0	69938.1
10	3	824	91	75	10	6791.1	496.5	56218.8
10	4	908	54	35	3	5113.3	340.5	68951.9
10	5	906	57	32	5	3324.5	268.5	53317.8
10	6	950	26	22	2	372.5	178.0	3871.2
10	7	965	15	19	1	310.9	151.0	2785.4
10	8	986	7	7	0	95.1	77.0	89.9
10	10	1000	0	0	0	33.4	33.0	4.5
11	2	759	111	84	46	13680.2	857.0	94690.5
11	3	828	102	45	25	6696.1	617.0	55815.2
11	4	868	77	41	14	5638.7	483.0	47450.6
11	5	924	47	22	7	2136.6	350.5	31410.9
11	6	921	43	34	2	763.9	279.0	8670.8
11	7	950	33	15	2	793.9	184.0	13774.1
11	8	978	11	10	1	186.8	153.0	221.6
11	9	985	7	8	0	745.3	83.0	19224.5
11	11	1000	0	0	0	36.6	36.0	5.0
12	2	715	155	69	61	20906.2	1154.0	115203.1
12	3	815	109	45	31	11284.2	857.0	72984.7
12	4	859	87	41	13	8464.9	602.0	80871.7
12	5	889	63	36	12	4027.8	454.0	52024.0
12	6	909	62	23	6	2822.8	356.0	26295.9
12	7	940	36	19	5	1966.4	292.0	22543.3
12	8	964	25	10	1	377.1	192.0	3012.6
12	9	974	10	14	2	235.7	160.0	1493.1

12	10	990	4	6	0	110.3	87.0	153.9
12	12	1000	0	0	0	40.9	40.0	5.9
13	2	700	150	75	75	27949.2	1432.0	145050.7
13	3	795	111	54	40	16317.3	1047.0	107682.1
13	4	801	118	58	23	8451.3	767.0	82289.9
13	5	868	90	29	13	5394.4	590.0	49898.1
13	6	880	76	31	13	4518.9	487.0	81579.9
13	7	915	54	19	12	3733.3	352.0	37381.2
13	8	944	33	21	2	586.0	293.0	3308.9
13	9	968	19	12	1	779.4	196.0	8188.5
13	10	967	20	10	3	276.9	169.0	2277.0
13	11	983	11	5	1	134.6	93.0	696.4
13	13	1000	0	0	0	44.9	44	6.2
14	2	683	155	66	96	32967.3	1632.0	146131.6
14	3	768	136	50	46	18404.7	1204.5	109410.5
14	4	769	154	45	32	9413.4	974.0	67997.0
14	5	836	99	36	29	9393.0	770.5	69971.6
14	6	869	84	26	21	6010.1	587.0	62506.3
14	7	906	58	24	12	5871.3	496.0	71601.3
14	8	921	49	23	7	2833.8	359.0	38408.6
14	9	940	38	19	3	1814.3	293.0	23762.8
14	10	973	17	7	3	458.3	201.0	5491.3
14	11	975	14	7	4	244.7	181.0	755.7
14	12	991	7	2	0	206.4	100.0	2328.3
14	14	1000	0	0	0	48.7	48	6.2

Table A.3: Algorithm learning results with $len_{max} = 10$ over 1000 strongly-connected, random Moore machines with a given number of states, of which a certain number have unique outputs.

Examples		Learning Success/Failure				Learning Steps		
states	unique	SUCCESS	TERM	LIMIT	LONG	μ-steps	η-steps	σ-steps
3	3	1000	0	0	0	8.1	8.0	1.2
4	2	949	15	36	0	54.0	44.0	36.1
4	4	1000	0	0	0	11.3	11.0	1.6
5	2	889	29	80	2	441.8	121.0	5180.7
5	3	961	13	26	0	67.2	49.0	87.3
5	5	1000	0	0	0	14.7	15.0	2.1
6	2	904	39	56	1	526.5	160.5	3434.7
6	3	919	28	53	0	152.3	119.0	173.0
6	4	970	13	17	0	75.0	56.0	98.5
6	6	1000	0	0	0	18.3	18.0	2.4
7	2	851	51	95	3	4403.0	284.0	78153.5
7	3	943	22	32	3	816.4	160.0	10279.0
7	4	938	19	43	0	205.0	122.0	1063.8
7	5	978	13	9	0	89.7	63.0	298.9
7	7	1000	0	0	0	22.0	22.0	2.8
8	2	864	46	79	11	9883.4	350.0	96995.5
8	3	888	54	58	0	3454.0	283.0	58562.2

8	4	930	28	37	5	439.0	168.0	5682.0
8	5	946	17	36	1	153.8	133.0	115.9
8	6	983	11	6	0	77.8	66.0	40.6
8	8	1000	0	0	0	25.4	25.0	3.3
9	2	813	83	86	18	7220.7	519.0	59390.3
9	3	886	54	56	4	3748.7	355.0	62660.2
9	4	901	48	48	3	493.9	276.0	1653.9
9	5	941	20	38	1	290.8	174.0	1159.5
9	6	951	18	28	3	174.2	139.0	238.2
9	7	990	5	5	0	102.8	73.0	298.2
9	9	1000	0	0	0	29.3	29.0	3.9
10	2	795	92	85	28	7029.9	628.0	69938.1
10	3	824	91	75	10	6791.2	496.5	56218.8
10	4	909	54	35	2	5115.9	340.5	68952.1
10	5	907	57	32	4	3324.5	268.5	53317.8
10	6	951	26	22	1	372.5	178.0	3871.2
10	7	965	15	19	1	310.9	151.0	2785.4
10	8	986	7	7	0	95.1	77.0	89.9
10	10	1000	0	0	0	33.4	33.0	4.5
11	2	752	118	84	46	11921.0	851.0	84526.8
11	3	822	109	45	24	6746.4	613.0	56050.6
11	4	863	84	41	12	5672.7	483.0	47632.4
11	5	918	53	22	7	2210.8	351.0	31691.4
11	6	915	48	34	3	769.7	277.0	8707.3
11	7	947	36	15	2	792.3	185.0	13778.1
11	8	975	12	10	3	190.4	153.0	247.2
11	9	984	7	8	1	133.1	83.0	871.3
11	11	1000	0	0	0	36.6	36.0	5.0
12	2	700	173	72	55	19409.2	1138.5	110705.4
12	3	806	120	45	29	11782.0	857.0	68553.9
12	4	842	103	41	14	8600.5	598.5	81660.4
12	5	878	76	36	10	4047.7	451.5	52042.0
12	6	898	73	23	6	1626.3	356.0	13062.5
12	7	934	43	19	4	1919.6	291.0	24849.1
12	8	962	28	10	0	352.5	192.0	2910.8
12	9	968	17	14	1	235.4	161.0	1498.4
12	10	991	3	6	0	109.8	87.0	142.0
12	12	1000	0	0	0	40.9	40.0	5.9
13	2	682	176	75	67	25126.9	1410.0	135174.8
13	3	760	145	55	40	12466.9	1023.5	75514.6
13	4	775	148	58	19	7214.0	752.0	56161.1
13	5	847	111	30	12	4426.4	584.0	42466.8
13	6	863	91	32	14	6199.6	484.5	94792.1
13	7	902	69	19	10	5264.4	351.0	59586.7
13	8	936	39	21	4	667.2	291.0	4257.4
13	9	962	25	12	1	672.0	196.0	7166.8
13	10	966	23	11	0	278.4	169.5	2282.2
13	11	979	14	5	2	137.3	93.0	703.5
13	13	1000	0	0	0	44.9	44	6.2

14	2	640	202	66	92	32351.0	1538.5	141014.8
14	3	730	177	50	43	18741.7	1177.5	113230.5
14	4	732	194	45	29	9252.4	969.5	68722.7
14	5	793	140	36	31	9950.7	756.0	88847.2
14	6	834	116	29	21	3812.2	577.0	31517.4
14	7	880	87	24	9	5657.7	496.0	72711.1
14	8	907	64	23	6	3308.9	355.0	41287.3
14	9	929	48	19	4	2183.5	292.0	25813.8
14	10	959	33	7	1	469.7	201.0	5561.4
14	11	973	18	7	2	236.3	182.0	657.4
14	12	989	8	2	1	194.5	99.5	1949.2
14	14	1000	0	0	0	48.7	48	6.2

Table A.4: Learning results of L^* *with resets* over 1000 strongly-connected, random Moore machines with a given number of states, of which a certain number have unique outputs, for comparison.

Examples		Learning Success/Failure		Learning Steps		
states	unique	SUCCESS	LONG	μ-steps	η-steps	σ-steps
3	3	1000	0	36.0	34.0	2.0
4	2	1000	0	50.8	50.0	4.8
4	4	1000	0	51.9	50.0	2.8
5	2	999	1	74.0	70.0	15.1
5	3	1000	0	69.3	66.0	5.7
5	5	1000	0	69.8	70.0	4.8
6	2	999	1	92.0	86.0	15.7
6	3	997	3	91.3	86.0	15.3
6	4	999	1	87.8	86.0	6.8
6	6	1000	0	88.8	86.0	6.3
7	2	998	2	118.0	110.0	23.1
7	3	996	4	111.7	106.0	16.1
7	4	1000	0	110.9	106.0	14.5
7	5	1000	0	108.4	106.0	9.0
7	7	1000	0	109.2	106.0	8.5
8	2	988	12	137.5	130.0	26.1
8	3	1000	0	135.6	130.0	21.6
8	4	994	6	131.8	126.0	16.0
8	5	997	3	131.0	126.0	15.4
8	6	1000	0	129.9	126.0	11.2
8	8	1000	0	129.9	126.0	10.4
9	2	981	19	163.8	150.0	32.8
9	3	996	4	158.9	150.0	25.8
9	4	997	3	157.1	150.0	23.4
9	5	999	1	154.1	150.0	18.4
9	6	995	5	153.4	146.0	18.5
9	7	1000	0	151.8	146.0	13.3
9	9	1000	0	152.5	150.0	13.0
10	2	972	28	182.9	170.0	31.4
10	3	990	10	183.6	170.0	32.3

10	4	997	3	178.2	170.0	22.9
10	5	995	5	179.0	170.0	24.3
10	6	998	2	176.0	170.0	19.3
10	7	999	1	174.9	170.0	18.4
10	8	1000	0	175.0	170.0	15.9
10	10	1000	0	174.4	170.0	14.8
11	2	954	46	211.3	198.0	37.0
11	3	975	25	205.5	194.0	31.2
11	4	986	14	206.2	194.0	32.4
11	5	993	7	201.5	194.0	25.4
11	6	998	2	199.2	194.0	22.3
11	7	998	2	197.5	194.0	18.1
11	8	999	1	198.3	194.0	18.8
11	9	1000	0	198.5	194.0	18.3
11	11	1000	0	197.2	194.0	16.7
12	2	939	61	237.1	222.0	39.6
12	3	969	31	233.7	218.0	37.2
12	4	987	13	229.4	218.0	33.2
12	5	988	12	226.0	218.0	31.2
12	6	994	6	223.4	214.0	24.7
12	7	995	5	224.9	218.0	26.3
12	8	999	1	222.7	218.0	23.1
12	9	998	2	220.8	214.0	20.8
12	10	1000	0	222.2	218.0	19.5
13	2	925	75	261.1	242.0	43.7
13	3	960	40	257.7	242.0	41.5
13	4	977	23	256.2	242.0	37.3
13	5	987	13	252.9	242.0	32.7
13	6	987	13	250.5	242.0	30.5
13	7	988	12	250.3	242.0	27.6
13	8	998	2	249.0	242.0	26.7
13	9	999	1	246.4	242.0	23.2
13	10	997	3	246.8	242.0	24.5
13	11	999	1	247.0	242.0	21.6
14	2	904	96	290.1	270.0	50.7
14	3	954	46	284.3	266.0	45.9
14	4	968	32	280.9	266.0	41.0
14	5	971	29	279.2	266.0	38.6
14	6	979	21	276.2	266.0	33.7
14	7	988	12	276.0	266.0	32.3
14	8	993	7	275.6	266.0	31.6
14	9	997	3	271.6	266.0	27.9
14	10	997	3	270.7	262.0	25.9
14	11	996	4	272.8	266.0	27.7
14	12	1000	0	271.9	266.0	24.1

Bibliography

[1] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014. (Cited on page 60.)

[2] Bernhard K. Aichernig and Christian Burghard. Giving a model-based testing language a formal semantics via partial MAX-SAT. In Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak, editors, *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, volume 12543 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2020. (Cited on page 61.)

[3] Bernhard K. Aichernig, Christian Burghard, and Robert Korosec. Learning-based testing of an industrial measurement device. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2019. (Cited on pages 2, 37, 54, 60, 61 and 63.)

[4] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 74–100. Springer, 2018. (Cited on pages 1 and 59.)

[5] Bernhard K. Aichernig, Edi Muskardin, and Andrea Pferscher. Learning-based fuzzing of IoT message brokers. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, pages 47–58. IEEE, 2021. (Cited on page 63.)

[6] Bernhard K. Aichernig, Andrea Pferscher, and Martin Tappler. From passive to active: Learning timed automata efficiently. In Ritchie Lee, Susmit Jha, and Anastasia Mavridou, editors, *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*, volume 12229 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2020. (Cited on page 63.)

[7] Bernhard K. Aichernig and Martin Tappler. Efficient active automata learning via mutation testing. *Journal of Automated Reasoning*, 63(4):1103–1134, 2019. (Cited on page 61.)

[8] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. Benchmarking combinations of learning and testing algorithms for active automata learning. In Wolfgang Ahrendt and Heike Wehrheim, editors, *Tests and Proofs - 14th International Conference, TAP@STAF 2020, Bergen, Norway, June 22-23, 2020, Proceedings*, volume 12165 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2020. (Cited on pages 59, 61 and 62.)

[9] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. (Cited on page 41.)

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. (Cited on pages 1, 2, 4, 10, 45, 59, 60 and 61.)

[11] AVL List GmbH. *AVL 415SE Smoke Meter - product guide*, Sep. 2013. (Cited on pages 1, 2, 52, 54, 61 and 62.)

[12] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, pages 200–236, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (Cited on page 41.)

[13] Nicolas Brémond and Roland Groz. Case studies in learning models and testing without reset. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, pages 40–45, 2019. (Cited on page 60.)

[14] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer, 1994. (Cited on page 59.)

[15] Yingke Chen, Hua Mao, Manfred Jaeger, Thomas Dyhre Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning Markov models for stationary system behaviors. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 216–230, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cited on page 60.)

[16] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978. (Cited on pages 6, 13 and 61.)

[17] Python Software Foundation. Python language reference, version 3.10. `https://www.python.org`, 2022. Last accessed on May 19, 2022. (Cited on pages 1, 2, 45 and 59.)

[18] Georgios Giantamidis, Stavros Tripakis, and Stylianos Basagiannis. Learning Moore machines from input–output traces. *International Journal on Software Tools for Technology Transfer*, 23(1):1–29, 2021. (Cited on page 7.)

[19] Roland Groz, Nicolas Brémond, Adenilso Simao, and Catherine Oriat. hW-inference: A heuristic approach to retrieve models through black box testing. *Journal of Systems and Software*, 159:110426, 2020. (Cited on pages 1, 2, 60 and 63.)

[20] Roland Groz, Catherine Oriat, and Nicolas Brémond. Inferring non-resettable Mealy machines with n states. In *International Conference on Grammatical Inference*, pages 30–41. PMLR, 2017. (Cited on page 9.)

[21] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing. (Cited on pages 2 and 59.)

[22] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing. (Cited on pages 15 and 59.)

[23] Kurt Jogun. A universal interface for the integration of emissions testing equipment into engine testing automation systems: The VDA-AK SAMT-interface. Technical report, SAE Technical Paper, 1994. (Cited on page 52.)

[24] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, 1994. (Cited on pages 2 and 59.)

[25] A.S. Klimovich and V.V. Solov'ev. Transformation of a Mealy finite-state machine into a Moore finite-state machine by splitting internal states. *Journal of Computer and Systems Sciences International*, 49(6):900–908, 2010. (Cited on page 7.)

[26] Zvi Kohavi and Niraj K. Jha. *Switching and finite automata theory*. Cambridge University Press, 3rd edition, 2009. (Cited on page 9.)

[27] Edward F. Moore. Gedanken-experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies. (AM-34), Volume 34*, pages 129–154. Princeton University Press, 2016. (Cited on page 6.)

[28] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: An active automata learning library. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, Australia, October 18-22, 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021. (Cited on pages 1, 2, 15, 45 and 59.)

[29] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012. (Cited on page 4.)

[30] José Oncina and Pedro García. *Inferring regular languages in polynomial update time*, volume Volume 5 of *Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, Feb 1993. (Cited on page 59.)

[31] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002. (Cited on page 1.)

[32] Andrea Pferscher and Bernhard K. Aichernig. Learning abstracted non-deterministic finite state machines. In Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak, editors, *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, volume 12543 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2020. (Cited on page 63.)

[33] Davy Preuveneers, Wouter Joosen, and Elisabeth Ilie-Zudor. Robust digital twin compositions for Industry 4.0 smart manufacturing systems. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 69–78, 2018. (Cited on page 55.)

[34] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993. (Cited on pages 1, 2, 8, 60 and 62.)

[35] Claude E. Shannon. Communication in the presence of noise. *Proceedings of the IEEE*, 86(2):447–457, 1998. (Cited on page 41.)

[36] Fei Tao, He Zhang, Ang Liu, and A. Y. C. Nee. Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2019. (Cited on page 55.)

[37] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn – learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems*, pages 216–235, Cham, 2019. Springer International Publishing. (Cited on page 63.)

[38] M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, Jul 1973. (Cited on page 9.)

[39] Felix Wallner. w-moore – reset-less active automata learning algorithm for Moore machines developed in Python with the AALpy framework. `https://gitlab.com/felixwallner/w-moore`, 2022. Last accessed on May 19, 2022. (Cited on pages 2, 45 and 59.)